NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

COMPLEXITY MEASURE FOR THE PROTOTYPE SYSTEM DESCRIPTION LANGUAGE (PSDL)

by

Joseph P. Dupont

June 2002

Thesis Advisor: Valdis Berzins
Co-Advisor: Michael R. Murrah

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2002	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Complexity Measure for the Prototype System Description Language (PSDL)	**5. FUNDING NUMBERS**
6. AUTHOR(S) Joseph P. Dupont	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(maximum 200 words)*

We often misunderstand, ill define or improperly measure the complexity of software. Software complexity is represented by the degree of complication of a system determined by such factors as control flow, information flow, the degree of nesting, the types of data structures, and other system characteristics, such as unconventional architectures. However, a common notion of software complexity fulfills a non-functional requirement, that of understandability. How well do we understand the control flow, the data structure, etc?

Rapid prototyping is an excellent tool to define system requirements and decrease developmental risk. Software complexity measured early (i.e., during prototyping), helps to minimize the complexity, which in turn helps to decrease the developmental risk also. The Prototype System Description Language (PSDL) provides the necessary code to achieve rapid prototyping. As a result, we have a need to accurately measure the complexity of PSDL.

14. SUBJECT TERMS Complexity, Software Complexity, Software Complexity Measures, Measurement Theory, Scale, Scale Type, PSDL, CAPS, PSDL Complexity	**15. NUMBER OF PAGES** 171
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

COMPLEXITY MEASURE FOR THE PROTOTYPE SYSTEM DESCRIPTION LANGUAGE (PSDL)

Joseph P. Dupont
Major, United States Army
B.S. in Electrical Engineering, University of New Hampshire, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2002

Author:	Joseph P. Dupont
Approved by:	Valdis Berzins, Thesis Advisor
	Michael R. Murrah, Co-Advisor
	Luqi Chair, Department of Software Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

We often misunderstand, ill define or improperly measure the complexity of software. Software complexity is represented by the degree of complication of a system determined by such factors as control flow, information flow, the degree of nesting, the types of data structures, and other system characteristics, such as unconventional architectures. However, a common notion of software complexity fulfills a non-functional requirement, that of understandability. How well do we understand the control flow, the data structure, etc?

Rapid prototyping is an excellent tool to define system requirements and decrease developmental risk. Software complexity measured early (i.e., during prototyping), helps to minimize the complexity, which in turn helps to decrease the developmental risk also. The Prototype System Description Language (PSDL) provides the necessary code to achieve rapid prototyping. As a result, we have a need to accurately measure the complexity of PSDL.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS

ADT	Abstract Data Type
AO	Atomic Operator
CAPS	Computer Aided Prototyping System
CO	Composite Operator
D	Deadline
DF	Dataflow
DSI	Data Stream In
DSO	Data Stream Out
EG	Execution Guards
ES	External System
FGC	Fine Granular Complexity
ISC	Internal Software Component
L	Latency
LGC	Large Granular Complexity
LOC	Lines of Code
MCC	McCabe’s Cyclomatic Complexity
MCP	Minimum Calling Period
MET	Maximum Execution Time
MRT	Minimum Response Time
NL	No Latency
NSS	Non-State Stream
OO	Object Oriented
P	Period
PSDL	Prototype System Description Language
PT	Primitive Type
SLOC	Source Lines of Code
SS	State Stream

TA	Triggered by All
TS	Triggered by Some
ZD-MIS	Zuse/Drabe Measurement Information System

ACKNOWLEDGMENTS

Acknowledgements often go overlooked but enough cannot be said to and about the people instrumental in writing and researching a thesis. I knew not what lay ahead as I embarked on this mission. I am a better person for doing it, or at least a little bit more educated.

I certainly have to thank my primary advisor, Major Michael Murrah who got me interested in this topic early after my arrival to the Naval Postgraduate School. Without the constant prodding to assist him in his dissertation research, I probably would still be looking for a thesis topic. The list of support he provided is endless and a simple word of thanks is not enough.

Secondly, multiple displays of encouragement were always available from Dr. Mantak Shing and Prof. Richard Riehle, the two people who make the Software Engineering Program first class. Dr. Shing with his never-ending knowledge of Prototype System Description Language (PSDL) and Computer Aided Prototyping System (CAPS) was always a constant source of reference and Prof. Riehle with his never-ending knowledge of software engineering took the time to push me through this demanding program.

I saved the best, for last. No words can express the gratitude I have towards my wife, Kit. Through illness and caring for our children, she always ensured I had the time I needed for my studies. Without her, this would not have been possible.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. AREA OF RESEARCH

This thesis provides an alternative measure of software complexity for the Prototype System Description Language (PSDL). Alternative complexity measures provide direct application for improvement to Dr. Juan Carlos Nogueira's [Ref. 10] Software Risk Model, and to Major Michael Murrah's dissertation research, a Modified Risk Model.

B. RESEARCH QUESTIONS

1. Principal Research Question

Does the software complexity measure (Large Granular Complexity and Fine Granular Complexity), as implemented in Nogueira's Software Risk Model [Ref. 10], accurately reflect the complexity of PSDL code?

2. Secondary Research Questions

- What is the definition of software complexity and how is it applicable to PSDL?
- Can the software complexity of a project be accurately determined during the prototyping stage?
- Is there a hybrid measure that more accurately represents the software complexity of PSDL code?
- What is a measurement?

C. DISCUSSION

Software has forever changed our lives. It is very difficult to find any system, mechanical, electrical or otherwise that does not run on software. Software has become an integral part of everything we see or do. Often times we are not even aware of the software because we cannot interact with it – embedded software. Many times the software is critical to the safety of the system – safety critical software. Other times the software must perform under precise timing constraints and work concurrently with other functions – real time software. Moreover, we no longer just build single systems that run on software; we build systems of systems. No longer, are we concerned with the software controlling a particular system; we have to be concerned with how system independent software interacts with other system independent software.

Our world is now defined by ever-increasing complex systems. By complex, we mean multifaceted, not difficult. System engineering helps us to simplify multifaceted systems ensuring a level of understanding. Decades ago it would not have been possible to build some of the systems of today. Technology did not allow it, and our processes did not allow it. Complex, then, meant difficult. We have difficult systems to build today, too, because of technology and our processes. They will not be complex (i.e., difficult) tomorrow, or will they still be complex (i.e., multifaceted)?

Whatever definition we choose, all software has inherent complexity. Be it multifaceted like a composition of many programs or be it difficult like a new programming language, complexity will always mean the unlikelihood of understanding. System engineering helps to minimize the complexity of multifaceted systems. What do we have to minimize the complexity of systems that are difficult?

Being able to produce a reliable measure of the eventual complexity of the software early in the design phase (i.e., during prototyping) is of considerable interest. Early (rather than later) in the development cycle, a software designer has the greatest flexibility in modifying the software design to achieve desired program objectives of cost, time, and functionality. Thus, investigating how and when to obtain early measures of complexity is of significant importance. Keeping complexity small can guarantee less risk, less error, and better maintainability.

D. SCOPE OF THESIS

The scope of the thesis will include the following:

- Examine some existing complexity measures and determine their applicability when applied to the Computer Aided Prototyping System (CAPS) environment.
- Exercise analyzer tools that will calculate side-by-side complexity measures (i.e. Nogueira's Large Grain Complexity, McCabe's Cyclomatic Complexity).
- Compare the various complexity measures of several available PSDL models.

- Derive conclusions regarding what complexity measures are best suited for the CAPS environment to include a possible hybrid measure.

E. METHODOLOGY

The methodology used in this research consists of the following steps:

- Exercise analyzer tools to parse PSDL code to facilitate comparison and analysis of various complexity measures.
- Conduct statistical analysis on the various results.
- Conduct a literature search of books, magazine articles, World Wide Web, and other library information resources regarding the definitions and measures of software complexity. Identify the ideal characteristics of a complexity measure for PSDL.
- Draw any conclusions on the various measures.
- Derive a PSDL complexity measure that is an accurate representation of the complexity of the final software system.

F. ORGANIZATION

This thesis is written for two readers: those who are familiar with CAPS and PSDL and those who are not. For those unfamiliar, it is recommended the reader first go through Appendix A. Although Appendix A is referenced frequently throughout the thesis, in its entirety it provides a comprehensive understanding of the CAPS design process and initial complexity analysis.

With an understanding of CAPS and PSDL, a reader can then go straight to Chapter V for the complexity measurement without having to read Chapters II - IV. For those who wish to gain an appreciation on how the measurement was derived, the reader should take the time to go through each chapter and appendix. The remainder of this thesis is divided into the following:

- Chapter II: *Software Complexity Defined.* Chapter II provides a definition of software complexity. It explores the importance of measuring software complexity.
- Chapter III: *Software Complexity Measures.* Chapter III provides a definition of existing software complexity measures. It explores the history and theory of measurements and important axioms related to software complexity measures.
- Chapter IV: *PSDL Characteristics.* Chapter IV provides a definition of PSDL and important factors necessary in measuring its complexity. It explores the semantics behind PSDL.

- Chapter V: *PSDL Complexity Measure.* Chapter V provides a definition of the PSDL Complexity Measure. It explores the development of the measure, through hybrids, information flow, PSDL properties and important weighting factors associated with those properties.
- Chapter VI: *Future Research and Considerations.* Chapter VI provides thoughts for future research and considerations that could not be part of this thesis due to timing constraints. Some of these are simply my own ideas others are questions that surfaced toward the end of my work.
- Appendix A: *Complexity Metrics for DCAPS.* Appendix A is also listed as Ref. 2 in the List of References. It is the final report for the SW 4510 class. Its importance is it became the catalyst that launched this thesis. It is the initial research and analysis conducted into deriving a complexity measure for PSDL.
- Appendix B: *PSDL and the Axioms of Chapter III.* Appendix B contains a reference of each axiom from Chapter III and its relation to PSDL. This appendix can be used after a sufficient understanding of CAPS and PSDL has been achieved.
- Appendix C: *PSDL Source File for Autopilot Control System.* Appendix C contains source code used to calculate the complexity of the Autopilot Control System. This system was used as the example in Chapter V.

G. BENEFIT OF THE STUDY

CAPS is an evolving prototyping system built and maintained by students and faculty of the Naval Postgraduate School. Accurately capturing the complexity of its generated PSDL code will provide critical insight into further development of both CAPS and Software Risk Models. This level of detail provides benefit to the Department of Defense, program managers of software projects as well as their developers.

II. SOFTWARE COMPLEXITY DEFINED

A. OVERVIEW

Providing clarity to software complexity is complex in and of itself because software is inherently complex (i.e., difficult, vague). Furthermore, the definition of complexity is inherently complex (i.e., diverse, compound). Therefore providing clarity to software complexity is difficult, vague, diverse and compound, a complex complexity!

The term, complexity, is an abstraction, a polymorphism that only needs specification to understand its semantics and quell its ambiguity. To establish specification of software complexity, we must consider what software is, the value and types of software measures, and finally explore the multiple ways to represent software complexity.

B. A MODEL OF THE WORLD

> The creation of software is difficult primarily because software is essentially a model of the real or conceptual world. Any such world is filled with a complexity that exceeds the capabilities of any one person to completely comprehend at one time. [Ref. 1]

Other engineering disciplines sometimes refute software engineering because the engineering of software follows no widely accepted methods or practices. Software, being intangible and a model of the world, exists outside the physical world where most engineering sciences lie. One of the most important arguments refuting engineering status is the lack of being able to measure software. After all, how do you measure something that is intangible? What do you measure? A major difference between a "well developed" science such as physics and some of the less "well developed" sciences such as psychology or sociology is the degree to which things are measured [Ref. 12]. The physicist, Lord Kelvin (1824-1904), is quoted as saying:

> When you can measure what you are speaking about, and express it into numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.

In software development, it is important to be able to measure the complexity of software to minimize risk in the development process. Risk minimization occurs because the decisions made from the quantitative understanding of the program help achieve better quality. However, how do you measure if one program is more complex or equally complex to another? What is the complexity of a program? What does the term complexity mean in general?

C. COMPLEXITY MEASURES

All software programs have inherent complexity, the more complex the program, the greater likelihood of misunderstanding. For the developer this means a greater chance of error during development, integration or maintenance. For the user it means greater chance of human error during installation or use. For all stakeholders, it means greater risk. Complexity, therefore, personifies several definitions.

Thomas McCabe [Ref. 7] developed McCabe Cyclomatic Complexity (MCC), which uses the number of independent paths through a program to describe complexity. Maurice Halstead [Ref. 3] considers unique operators and operands. Literal definitions include a simple level of understanding or readability of the code and others consider the total Lines of Code (LOC) or “Software Size”. Function Point analysis uses a weighting system of the number of inputs, outputs, queries, files, and system interfaces required in the program. In fact, research shows that there are as many as 100 different ways to measure complexity [Ref. 14], each with its own interpretation. With so many complexity measures to choose from, the question becomes what is the appropriate measure for your situation, or your code.

Considering “Software Size” alone yields numerous definitions – for instance, Whitmire [Ref. 13] as paraphrased by Pressman [Ref. 11] identifies four different views regarding what size means in Object Oriented programming:

> Size is defined in terms of four views: population, volume, length, and functionality. Population is measured by taking a static count of object oriented (OO) entities such as classes or operations. Volume measures are identical to population measures but are collected dynamically, -- at a given instant of time. Length is a measure of a chain of interconnected design elements (e.g., the depth of an inheritance tree is a measure of

length). Functionality metrics provide an indirect indication of the value delivered to the customer by an OO application.

While software size and complexity have been extensively researched, there are still no conclusive software complexity measures that can be captured very early in the software development lifecycle (perhaps during prototyping) that produce reliable estimates of the eventual complexity of the delivered software. Popular methods, such as Function Point analysis and Halstead's Complexity measure have several weaknesses because they were developed in the seventies and do not reflect the intricacies of OO programming.

Dr. Nogueira [Ref. 10] performed one initial investigation of an early calculable complexity measure for the PSDL). He used this measure as an input to his software risk model. However, questions remain about this measure because there limited documentation exists explaining the logic behind his choice.

Being able to produce a reliable measure of the eventual complexity of the software early in the software's design is of considerable interest. Early rather than later, in the development cycle, a software designer has the greatest flexibility in modifying the software design to achieve desired program objectives of cost, time, and functionality. Thus, investigating how and when to obtain early measures of complexity is of significant importance. Whatever the definition, keeping complexity manageable can guarantee less risk, less error, and better maintainability.

D. THE ILL-DEFINITION

To illustrate the ill definition of complexity I will use two examples. Figure 2.1 provides the first example using extracts of Java code.

P1

```
for(int i=1, i <= 100, i ++)
{
       sum += i;
}
```

P3

```
int i=1;
for (i <=100)
{
       sum = sum + i;
       i = i + 1;
}
```

P2

```
for(int i=1, i <= 100, i ++)
{
      sum = sum + i;
}
```

P4

```
int i = 100;
int sum = i * ((i + 1)/2);
```

Figure 2.1. Program Complexity Using Java.

Intuitively you can easily see that P1, P2 and P3 add all the numbers 1 - 100 recursively. Syntactically there is a little difference but the semantics are exactly the same. The syntax of P4 is different, though. However, its semantics is also the same as the other three. Certainly is not as intuitive by looking at the code, unless you have a good math background. One, therefore, can say that P4 is the most complex because it is more difficult to understand or is not readily understandable by looking at the code. You can add comments to P4 to make it more understandable for another programmer to maintain the code. Except, depending on an individual's viewpoint, commented code may add to instead of detract from the total complexity because you are adding more lines to it, more information.

When LOC defines complexity, which usually refers to logical source lines of code, P3 is the most complex leaving P1, P2 and P4 equivalent (i.e., P3 > P1 = P2 = P4). Nevertheless, my viewpoint tells me P3 is not the most complex because I understand what it is doing. Quite honestly, I could argue that P3 is the least complex. P3 lays out all the instructions, nicely, making the code more easily read and understood without comments. Therefore, without specifying a definition of complexity you cannot determine the complexity. It becomes important to understand the context in which we speak of complexity.

The second example is a quick glance at the potential problem of trying to define the complexity of PSDL. If complexity had a simple definition there should be correlation between the many different measures. In other words, a one to one relationship should exist. A program should show signs of being more complex using any measure. Although, Figure 2.1 briefly showed this is not the case. Without doubt, if all measures were created equal, you should not expect one measure to show greater complexity and another to show a lesser degree of complexity. In Figure 2.2 a number of PSDL programs were analyzed using LOC, MCC and Noguiera's Large Granular Complexity (LGC) measures.

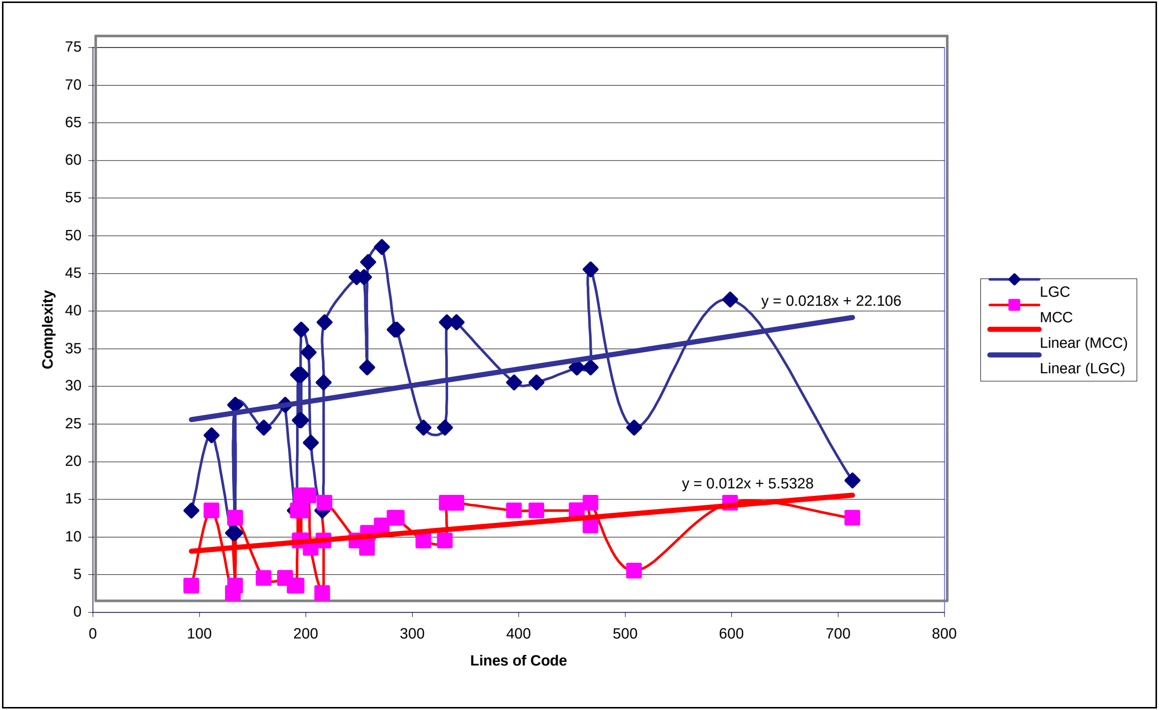


Figure 2.2. LGC and MCC Plotted against LOC. "From [Ref. 2]"

Figure 2.2 is a chart of three different software measures that can be used to calculate software complexity. The detail of these measures is provided in Appendix A, [Ref. 2] and is not needed here to discuss their relationship. LGC and MCC are each plotted on the y-axis against the third measure, LOC, on the x-axis. LGC is the top set of points and MCC, the bottom set of points. LOC in PSDL is uncommented source lines of code.

The trend lines in Figure 2.2 give you the impression that there is a linear relationship between lines of code and the complexity measures. Nogueira's LGC measure and MCC measure increase as LOC increases. However, we have already shown in Figure 2.1 where LOC may not accurately represent complexity so this may be a flawed theory. Furthermore, because the slope of the MCC trend line (i.e., 0.012) is roughly one half that of LGC (i.e., 0.0218), we see that MCC is more independent of LOC than that of LGC. The similar peaks and valleys of each LGC and MCC point show a relationship between themselves. Yet, there are some instances where the LGC increased and the MCC did not. The degree of change also does not match up.

The importance of these findings is two-fold: complexity measures do **differ** and the **choice** of measure is important. Once the proper measure is chosen, you must understand the meaning behind the numbers. A number by itself has no meaning.

A measurement based on some known formal standard has meaning. To say a board is three feet long presents a picture in our mind of a yardstick and a good idea of the actual length. But to say the board is a meter long, for some, presents no image and no understanding of its meaning. More specifically, in terms of software, twenty years ago to say a program had 100,000 lines of code would conjure up images of a large, complex program that took many man-months to develop. Today that measurement is indicative of a fairly small program. So then, how do we choose the proper complexity measure that has and will have meaning?

E. SUMMARY

Software complexity takes on many different meanings. It truly is "In the eyes of the beholder." Because we all see complexity differently, the past three decades has given us well over 100 ways to measure the complexity of the software; each has its own interpretation. Most of the earliest complexity measures considered software size, either as volume or as LOC, as the most accurate way to measure complexity of software.

Unfortunately, we have several methods of determining that size. We can count independent paths through the code, unique operators and operands, or we can simply count the lines of code? On the other hand, if we decide to count lines of code we need to define what is actually counted (i.e., comments, functions, operations, etc).

In determining how to measure software complexity, some decisions have to be made beforehand. The independent evaluator must understand complexity's meaning. Understanding is best, when coupled by intuition. There has to be a precise definition that specifies the conditions of the measurement. The choice of programming language most certainly is a factor and more specifically the structure of the language. Lastly, it must be decided when in the lifecycle of software development the complexity will be measured.

Complexity measured early and minimized will surely benefit the program by minimizing the risk involved. It is unlikely there will ever be a single way to measure complexity, but a set of rules upon which to make those decisions is attainable.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SOFTWARE COMPLEXITY MEASURES

A. OVERVIEW

At the start of the 17th century, there was no way to quantify heat, hence, no way to measure temperature. Rules upon which to base the proper method of measuring encompass **measurement theory**, the ideas behind **scales and scale types**, and accepted **axioms**. One accepted **axiom**, regarding temperature, was that something always felt warm or cool. This axiom is fairly pragmatic but nonetheless the assumption had to be made to continue.

A temperature **scale** could not be arranged until minimum and maximum numbers were given to the temperature of some physical object. Those numbers then had intuitive significance and value. For instance, a number could be placed on the freezing and boiling points of water, because they are easily observed. In degrees Celsius the scale became 0° and 100°. In terms of temperature, these numbers have new meaning, and can be mapped back to an intuitive understanding of temperature, (i.e., cold and hot). The **scale type** defines the relationship of any two numbers within the scale and the mapping of values to an intuitive understanding is covered under **measurement theory**.

In the 20th century we found ourselves trying to quantify software complexity and therefore needed a way to measure it. Many measures have been derived because of and **categorized** by, certain program conditions, expectations and intuitive understandings. Software complexity measures are not exempt from the rules of measurement; consequently, it is important to understand those rules in detail and the categories in which they are placed.

B. MEASUREMENT THEORY

Measurements have empirical and formal relationships. In the world of empirical relationships, we must establish certain axioms for measurement to have meaning. These axioms are subjective, based on individual judgment, preferences or reactions. Put another way, for measurements to be effective there has to be an intuitive meaning. All axioms follow the laws of symmetry and transitivity, presented here as Axioms 3.1 and 3.2:

Axiom 3.1: If you prefer a to b, you do not prefer b to a.

Axiom 3.2: If you do not prefer a to b, and do not prefer b to c, then you do not prefer a to c.

These axioms can be prescriptive, based on some conditions of rationality, or descriptive based on some conditions of behavior that allow the measurement to take place. [Ref. 14]

Using a wooden board as an example, to say the board is three feet long is descriptive since we have an understanding of the length of a foot (i.e., a ruler) and a yard (i.e., a yardstick). It is prescriptive in the sense that length is a widely accepted measurement for something such as a board. Any rational person would not provide you a measurement of 10 lbs when speaking of a board. It would have no meaning especially when using that board on some construction project.

Formal relationships base measurements on widely accepted scales to have an intuitive meaning. A yard means 3 feet; 12 inches means a foot and we understand that 8 feet is the standard length of boards. Problems occur in measurement when the scale is unknown, such as the metric system that the U.S. has not fully adopted. Building a formal relational system is the process of putting numeric value to an empirical relation. To illustrate empirical relations versus formal relations we will use Figure 3.1.

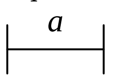
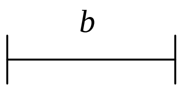
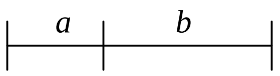
Empirical Relational System		**Formal Relational System**
Wooden Boards		□ Real Numbers
Relation: equal or longer than: · ³		**Relation:** equal or greater than: ³
		some measurement: *m*
		
b · ³ *a*	⇔	*m(b)* ³ *m(a)*
Binary Operation:		**Binary Operation:**
Concatenation or Composition (◦)		Addition (+)
		
a◦ *b* = *c*	⇔	*m(c)* = *m(a)* + *m(b)*

Figure 3.1. Empirical vs. Formal Relational System. "From [Ref. 14]"

1. Relations

In Figure 3.1, the **empirical** relationship of two wooden boards is interpreted as one board being equal or <u>longer</u> than another, using the symbol · ³. Whereas the **formal** relationship is one board being equal or <u>greater</u> than another, using the symbol ³. Hence *b* is longer than *a* if and only if the measurement of *b* (i.e., *m(b)*) is greater than the measurement of *a* (i.e., *m(a)*).

2. Binary Operations

A binary operation in **empirical** terms is represented by <u>concatenation</u> or <u>composition</u>. It simply means when *b* is placed end to end with *a*, we say that board *b* is concatenated to board *a* and that some new board *c* is the composition of board *a* and board *b* (i.e., *a*◦ *b* = *c*). On the other hand, in **formal** terms that binary operation is represented as the <u>sum</u> of two elements, *a* + *b*. As a result, *c* is the composition of *a* and *b* if and only if the measurement of *c* (i.e., *m(c)*) equals the sum of measurement *a* and measurement *b* (i.e., *m(a)* + *m(b)*).

This is a fairly explicit explanation to a very simple perception; the formal relationship has intuitive meaning and the empirical relationship maps to some formal measurement. Zuse [Ref. 15] calls this mapping a homomorphism. We must have a homomorphism to have a good measure and for a homomorphism to exist there must be an injective relationship between empirical and formal relational systems. With boards, we know the empirical relation is length; intuitively we know the meaning of length. The length of boards can be measured and the value of the measurement has one and only one meaning.

3. Theory of Software Complexity Measures

Measuring software complexity is somewhat more difficult because of an unknown empirical relation? When there is no intuitive understanding of complexity we cannot say reliably that one program is more or equally complex to another. Moreover, without an empirical relation we cannot intuitively predict that one program is more complex than another. Determining this empirical relation lies heavily with the individual trying to interpret the complexity because there is no standard interpretation of complexity, as there is with length. As long as the individual's understanding follows the axioms of measurement and accepted conditions of complexity, the measurement will hold true.

Some people wish to say complexity can be measured using LOC, which is nothing more than measuring the length of a program. If the length of a program defines an intuitive understanding of complexity, then it is valid. However, in Chapter II, we showed how there are some inaccuracies with this measurement based on other interpretations of complexity.

C. SCALE / SCALE TYPE

Before an ability to measure temperature existed, a scale, scale type and accepted axioms had to be established, not to mention having a tool to actually measure it. Axioms fell under the pretext that something felt warmer or cooler, that some things always had the same temperature, (i.e., freezing and boiling points of water, and the temperature of humans) and that there should be a way to measure the difference with standard values. We are in a similar position today trying to measure software complexity. Complexity measures, too, must have a defined scale, scale type, and axioms to provide us with useful data. It is important to distinguish between scale and scale type.

1. Explanation of Scale and Scale Type

Zuse [Ref. 15] indicates a scale is a homomorphic mapping; the formal relation must correlate with the empirical understanding of the measure. Scale types are defined by admissible transformations, in other words, what operations or deductions do we allow on the scale. There is a classification of scales according to their admissible transformations. The common hierarchy of scales is: the **nominal scale** with any one to one transformation, the **ordinal scale** with a strictly increasing transformation $g(x) = b,\ a = 0$, the **interval scale** with the transformation $g(x) = ax + b,\ a > 0$, the **ratio scale** with $g(x) = ax,\ a > 0$, and the highest on the hierarchy being the **absolute scale** with admissible transformation $g(x) = x$; that means that no admissible transformations of x can occur. [Ref. 15]

Knowing the proper scale type upfront allows you to manipulate the numbers correctly and provide meaningful data. It allows you to properly scale the measurements. Choosing the proper scale type is a subjective matter dependent upon how the individual wishes to represent their measurements. Table 3.1 shows an overview of each scale type followed by comprehensive definitions.

SCALE TYPE	**EXAMPLE**	**DESCRIPTION**	**APPLICABILITY TO COMPLEXITY**
nominal	set of license plates	one plate can only belong to one car	weak
ordinal	military rank	E1-E9, O1-O10 clearly indicates ranking order	strong – the basis
interval	Fahrenheit/Celsius	one unit on the scale represents the same magnitude across the whole range of the scale	weak
ratio	money	so many Deutche marks is equivalent to US dollars	strong
absolute	Kelvin	0^0K is the temperature where atoms have no movement and give off no heat	some

Table 3.1. Scale Types. "After [Ref. 15]"

[Ref. 5] provides the following definitions on scale types:

Definition 3.1: **Nominal.** Nominal variables allow for only qualitative classification. That is, they can be measured only in terms of whether the individual items belong to some distinctively different categories. We cannot quantify or even rank order those categories. For example, all we can say of two license plates is only one can belong to any given car and the difference between the two is that they represent different cars. You know nothing of the cars themselves and therefore cannot quantify the difference. Another example, two individuals are different in terms of some variable A (e.g., they are of different race), but we cannot say which one "has more" of the quality represented by the variable. Other examples of nominal variables are gender, color, city, etc.

Definition 3.2: **Ordinal.** Ordinal variables allow us to rank order the items we measure in terms of which has less and which has more of the quality represented by the variable, but still they do not allow us to say how much more. Using military rank as an example, we easily see an order of responsibility given to individuals based on their rank. However, we cannot say that every O4 has an equitable amount of responsibility and that an O2 has half as much responsibility. Another typical example of an ordinal variable is the socioeconomic status of families. For example, we know that upper-middle is higher than middle but we cannot say that it is, for example, 18% higher. In addition, the distinction between scale types (i.e., nominal, ordinal, interval, etc), itself, represents a good example of an ordinal variable. For example, we can say that nominal measurement provides less information than ordinal measurement, but we cannot say "how much less" or how this difference compares to the difference between ordinal and interval scales.

Definition 3.3: **Interval.** Interval variables allow us not only to rank order the items that are measured, but also to quantify and compare the sizes of differences between them. For example, temperature, as measured in the degrees of Fahrenheit or Celsius, constitutes an interval scale. We can say that a temperature of 40 degrees is higher than a temperature of 30 degrees, and that an increase from 20 to 40 degrees is the same as the increase from

40 to 60 degrees. What we cannot say is that 40 degrees is twice as warm as 20 degrees.

Definition 3.4: **Ratio.** Ratio variables are very similar to interval variables; in addition to all the properties of interval variables, they allow for statements such as x is two times more than y and still maintain the one to one relationship of the original measure. Typical examples of ratio scales are measures of money, time or length. For example, the US dollar represents and converts to other currencies dependent on the current rate of conversion. One inch will always equal 2.54 cm and two inches equal 5.08 cm, etc. Interval scales do not have the ratio property.

Definition 3.5: **Absolute.** Zuse [Ref. 15] and Merriam-Webster Dictionary [Ref. 8] state, absolute variables are unconditional. Absolute zero on the Kelvin temperature scale represents the temperature at which there is no heat. Absolute variables contain some properties of the ratio scale. In terms of the Kelvin temperature scale we can say 100 degrees is one half 200 degrees. Percentage measures are absolute scale measures. For example:

$$NC = number\_of\_comments / LOC$$

$$number\_of\_comments = a\,LOC\ .$$

$$NC = a$$

The value of NC has no transformation and therefore is absolute.

2. Scale Type for Software Complexity Measures

Length is instinctively defined as the distance between two points or the duration of time. This definition helps to choose the proper scale type because it provides necessary distinction and necessary assumptions. Length can be measured using multiple units, but the scale remains the same; it is always measured using the ratio scale.

Measuring software complexity, on the other hand, poses some challenges because of its ill **definition** and unstated **assumptions**. We must define what we are measuring before we can go any further. Software complexity can mean how well we understand the code, how well it reads, or how easily it can be maintained [Ref. 14]. The user who is analyzing software complexity must decide which **definition** best fits his empirical understanding; this becomes part of the **axioms** necessary in choosing a proper measure and helps to incorporate the right scale type.

There is a high degree of subjectivity in choosing the **definition** of complexity. This problem becomes evident if the user has to rank the complexity of some programs and he/she has to explain the reasons for the ranking [Ref. 14]. In Figure 2.1 we showed this subjectivity with four extracts of Java code. The complexity of the extracts could be interpreted in several ways, separated by the user's definition. Only through the definition, can the user measure complexity properly and only through measuring, can the user rank the complexity of the extracts. Supplying quantitative results through measuring, to a qualitative entity, provides objectivity to the subjective nature behind the definitions. Incorporating the proper scale type provides additional meaning to the ranking.

What are we comfortable in accepting to be **fact**? We know the basis of software complexity is said to lie on an ordinal scale (See Table 3.1). An ordinal scale will only allow us to say one program is more complex than another. We cannot say to what degree the difference is. If we wish to say that one program is twice as complex as another, we must choose measurements that fit on a ratio scale.

D. AXIOMS OF SOFTWARE COMPLEXITY MEASURES

The discussion of axioms has been presented throughout this chapter, requires no further explanation and best concludes with a definition from Zuse [Ref. 15]:

Definition 3.6: **Axiom.** "Axioms are conditions or basic assumptions of reality. Axioms are mostly empirical, but technical ones are also possible. Axioms formulate certain empirical properties. The goal in software measurement is to figure out empirical laws about software development, software complexity, software maintainability, etc. The discovery of qualitative laws of software quality and software development is another goal of the formulation of axioms in the area of software measurement. Further goals of formulating axioms are to get a more precise terminology in the area of software measurement."

Most important under this section is the presentation of axioms associated with software complexity measures. The first of those axioms is a restatement of what we now know of scale types; the second, a characterization of PSDL; others are made available to us from other authors:

Axiom 3.3: Software complexity measures lie on an ordinal or ratio scale.

Multiple software complexity measures originated over the last three decades, each with desirable properties. What is most interesting is that many of these desirable properties differ, due to simple bias amongst the authors. Each author had their own definition of complexity, program conditions, expectations and intuitive understandings. Some authors provided new measures, while others provided modifications to old measures [Zuse 14]. Some of those desirable properties apply to PSDL and are given here as axioms. The relationship of each axiom to PSDL is provided in Appendix B and can be used as reference any time the reader has a sufficient understanding of CAPS and PSDL.

The following list of axioms is presented as an intuitive evaluation and serves as the basis of determining a proper complexity measure for PSDL. Before this list could be developed, PSDL complexity had to be defined. Complete discussion of this definition is provided in Chapter IV but is given here, as Axiom 3.4, for justification of this list.

Axiom 3.4: Complexity of PSDL, and the augmented graphs associated with it, is implicitly defined as its understandability versus its readability and maintainability.

This is not an exhaustive list of axioms. This subset is used only to expose the primary approach of developing a complexity measure for PSDL. That approach specifically pertains to properties of **flow graphs, graph theory and nested structures**. The CAPS environment is built upon **dataflow** diagrams that produce **graphs** of vertices and edges.

For most structured or sequential programs, intuitively, an accepted fact is that **nesting** yields <u>more</u> complexity. However, for PSDL we believe the opposite to be true. During a CAPS prototyping session, **nesting** substructures under a main structure provides another level of abstraction. This increases modularization and provides greater cohesion. More importantly, this belief follows the principles of system engineering in which complex systems are decomposed to the smallest level possible to increase understanding and hence decrease the complexity. This belief also follows the rules of heuristics and "Chunk Theory" [Ref. 9] that says humans can only understand 7 ± 2 entities at any given time. The axioms are now presented as:

1. Axioms from Tsai, Lopez, Rodriguez and Volovik. [Ref. 14]

These axioms originally applied to dataflow measures, but there are similar findings for measures based on **flow graphs** [Zuse 14].

Axiom 3.5: One of the most significant properties of a metric is to conform to intuition. Intuition applied to the objects being measured means that objects, which are seemingly more complex (from one's previous experience) should also be declared as more complex when the metric is applied. Objects that are about equal complexity should also measure about the same. The point is that some objects seem simpler to most people than other objects, and the metric should, in most cases, confirm to this observation.

This axiom holds for Measurement Theory, in Section B, as well.

Axiom 3.6: Another property of metrics going hand in hand with intuition is consistency (or monotony). In other words, if data structure x is a substructure of a data structure y, then $Complexity(x) \leq Complexity(y)$.

This is self-explanatory.

Axiom 3.7: The measure should measure the structure of data, not only the size of data. Structure of data tends to be stable during the design process, whereas size of data might not be known even during run-time.

This refers back to our discussion in Chapter I and II that software size is only one way to measure software and is often not appropriate to measure complexity.

Axiom 3.8: It should be possible to use the metric at a stage of the software design when not all of the decisions have been already made. Measuring a finished product to guide its design is of no use. It is too late. To support these properties, the metric should tolerate incomplete information.

Simply put, it is important for complexity measures to be used early in software development.

Axiom 3.9: The metric should have the property of automation. Given a data structure description in some formal language, it should be possible to produce a formal machine ready representation of the data structure. The representation seen can be used as an input to a program and the set of measurements can be produced. Resulting measurements themselves should have such a form that humans can easily interpret them, as well as being easily used as an input to some metric-based design support system. To provide automation, the metric should be based on some mathematical foundation.

This axiom lists the importance of three items for complexity measures: automation, simple interpretation and use, and mathematical foundation**.** Zuse [Ref. 14] also comments that:

…although it is important for measures to be based on some mathematical foundation, it must be noted that a measure which is based on solid mathematical foundation may not be an appropriate software complexity measure.

This statement is taken to mean that other properties other than math may be more appropriate.

Axiom 3.10: Most people cannot manipulate more than a small amount of information at the same time unless there are visual tools available to assist them. Therefore, it is of importance to be able to visualize the process of measurement. It should be easy to present a pictorial representation of the data object and to illustrate graphically the process of application of the metric to a particular data structure.

This axiom holds for the rules of heuristics and "Chunk Theory" [Ref. 9]. It also further clarifies Axiom 3.9 to say that the measure should be easy to understand and use.

2. Axioms by Weyuker [Ref. 14]

Table 3.2 is provided for clarity:

NOTATION	EXPLANATION
P, Q, R	Program Bodies
$P;Q$	Is composed of P and Q, some binary operation.
$m(P)$	Denotes the complexity of P, with respect to some hypothetical measure, and is always a non-negative number.
$m(Q)$	Denotes the complexity of P, with respect to some hypothetical measure, and is always a non-negative number.
It holds for any P and Q : $m(P) \leq m(Q)$ *or* $m(Q) \leq m(P)$ - Two programs are either equally complex or not equally complex.	

Table 3.2. Notation of Weyuker. "From [Ref. 14]"

Axiom 3.11: $(\exists P)(\exists Q)\left(P \equiv Q \wedge m(P) \neq m(Q)\right)$

> The intuition behind this property is that even though programs compute the same function, it is the details of the implementation that determine the complexity of the program, not the function being computed by the program.

This axiom's literal meaning is: There exists some *P* and *Q*, *P* and *Q* are similar programs and their complexities are not equal. The literal meaning is not accepted because it is simply false.

The quoted definition is more exact and is acceptable. Its meaning is interpreted as: two programs having similar functions may not be equally complex because there is more to a program than just the functions it performs.

Axiom 3.12: $(\forall P)(\forall Q)\ \left(m(P) \leq m(P;Q) \text{ and } m(Q) \leq m(P;Q)\right)$

> We believe that "monotonicity" is another fundamentally important property and it is difficult to imagine the sense in which a measure which fails to satisfy the monotonicity property is measuring complexity.

This axiom's literal meaning is: For every program *P* and *Q*, the complexity of *P* and the complexity of *Q* is always less than or equal to any program that is composed of *P* and *Q*. This is completely acceptable and defines our use for PSDL. Refer back to Measurement Theory, and Figure 3.1; the use of composition and concatenation were given.

The literal meaning of monotonicity is: having the property either of never increasing or of never decreasing as the values of the independent variable or the subscripts of the terms increase [Ref. 8]. The use of the word monotonicity is neither accepted for our purposes nor does it seem appropriate when speaking of programs.

Axiom 3.13: $(\forall P)(\forall Q)\ \left(m(P)+m(Q)\leq m(P;Q)\right)$

> The question is, given that the complexity of a program body should be no less than the complexities or each of its parts, can we make a stronger statement? For example, should the complexity of a program body be no less than the sum of the complexities of its components? Intuitively, in order to implement a program, each of its parts must be implemented.

This axiom's literal meaning is: For every program *P* and *Q*, the sum of complexity *P* and complexity *Q* is less than or equal to the complexity of a program that is composed of *P* and *Q*. In short, this entire axiom can mean that the whole is greater than the sum of its parts. Measurement Theory holds for this axiom because it relates the sum of complexities as being less than <u>or equal</u> to the complexity of a concatenated program.

3. McCabe Cyclomatic Complexity (MCC) Measure

McCabe has worked with complexity measures since the mid 1970's when he formulated his first measure based on **flow graph theory** represented as {*MCC-V(G)*} (this represents a hyphenated abbreviation for McCabe's Cyclomatic Complexity versus the difference between two variables *MCC* and *V(G)*). In Appendix A [Ref. 2] we find that cyclomatic complexity, *v(G)*, is defined for each module to be:

$e - n + 2$

where:

e = the number of edges and

n = the number of nodes in the control flow graph

Cyclomatic complexity is known as *v(G)*, where *v* refers to the cyclomatic number in **graph theory** and *G* indicates that the complexity is a function of the graph *G*. Cyclomatic complexity is a measure of the number of independent paths that exist in a strongly connected, undirected graph (i.e., a strongly connected graph is one in which each node is reachable from every other node). It is precisely the minimum number of paths in linear combination that can generate all possible paths through the module.

Normally, the cyclomatic number in **graph theory** is defined as *e - n + 1*. However, McCabe [Ref. 7] points out that program control-**flow graphs** are not strongly connected, but can become strongly connected when adding a "virtual-edge" to connect the exit node to the entry node. Thus, the cyclomatic complexity definition for program control-**flow graphs** is derived from the cyclomatic number formula by simply adding one to represent the contribution of the virtual edge.

In PSDL a similar theory holds. A virtual parent node, not shown in the drawing pane, but indicated in the tree pane of CAPS, is added to every dataflow diagram. The diagram is actually nested under this parent node. The two are virtually connected via a single edge. This node and edge remain constant for every diagram and receives no consideration as part of a PSDL complexity measure.

The axioms of **graph theory** are applicable to PSDL and are considered. McCabe also based his measurements on **graph theory**. Unfortunately, because his measurements lack sensitivity to **nesting**, they are not applicable to PSDL. Nonetheless, the presentation of McCabe's measures and axioms receive consideration because of his reputation with complexity measures and their significance and connection to **graph theory** rather than their direct applicability to PSDL. To keep consistent with the before mentioned axioms, discussion of this applicability is provided in Appendix B. The following is listed as a single axiom and is offered for its sagacity [Ref. 14] and [Ref. 7]:

Axiom 3.14:

- *MCC-V(G)*, is the maximum number of linearly independent paths in G; it is the size of a basis set.
- *MCC-V(G)* depends only on the decision structure of G. – The independent paths through the structure represent decision points for data to flow.
- *MCC-V(G) ³ 1*, there must be at least one independent path for every graph.
- G has only one path if and only if v(G) = 1

- Inserting or deleting functional statements to G does not affect v(G). – Only the independent paths affect G.
- Inserting a new edge in G increases v(G) by unity. – The addition of one edge increases the complexity by one.

E. CATEGORIES OF SOFTWARE COMPLEXITY MEASURES

In the previous section we presented axioms that represented a subset of desirable properties of software complexity measures. That subset also serves as the basis for deriving a complexity measure for PSDL. Axioms are conditions or basic assumptions of reality. Part of the reality with software complexity measures is that they fall under different categories based on their characteristics. This section helps to identify those categories.

Measurement is quantitative; it is about counting. What can be counted in software? Flow graphs are an important part of software engineering so we can count nodes, edges, and repetitions. In actual source code, we can count LOC, timing constraints, data types, etc. However, if we were to count LOC and define it as our complexity measure, we probably could not gather enough information to describe its understandability or readability but it may say something about its maintainability.

Choosing the right measure means understanding the individual characteristics of a particular measure, the individual characteristics of the software and the goals of the measurement. LOC is only one measurement and is representative of sizing measures. If complexity is interpreted as maintainability then knowing the length of a program may be useful.

1. Categories by Zuse [Ref. 15]

Size represents just one category under software measures. Zuse [Ref. 15] provides other classifications, as well. He specifically categorizes the following as software complexity measures, each represented with an example:

- Size Measures – LOC
- Data Structure Measures – Data processed by the program.
- Control Flow Measures – McCabe's Cyclomatic Complexity (MCC)
- Information Flow Measures – Henry et al.
- Software Science Measures – Halstead

Size is the notion that as things get larger or contain more parts, they are more difficult to work with and harder to understand. Something that is harder to understand is, therefore, more complex. As long as the user's instinct tells them size is understandability and understandability is complexity, it can be used properly under those conditions.

Data structures handle variables from the first time they are assigned to the last time they are referenced or assigned. If this is an important notion to the user, then it may be an important notion under complexity.

How a program **controls** the **flow** of data is another characteristic of programs that can be measured, and can be used to represent complexity under the right circumstances. MCC is a simple example of this type of measure.

Henry and Kafura [Ref. 4] gave us **information flow measures** in the early 80's. These measures are useful with larger programs containing multiple modules and describe how data flows between modules**.** They provide a different classification scheme of software measures, which is presented below.

Lastly, Halstead [Ref. 3] gave us some of the earliest software measures of the 70's that fit under the category of **software science measures**. Counting the unique number of operators and operands in a program, Halstead presents several equations to calculate program length, volume, level, purity and effort. Length and volume also fit under sizing measures.

2. Categories by Henry and Kafura [Ref. 4]

Henry and Kafura [Ref. 4] who gave us information flow measures, do not use size as a classification. They categorize software measures differently. The classification falls into the following three areas, each with experts providing measurements in that category:

- Lexical Content – Halstead, McCabe, Thayer
- Information Theoretic Concepts – Alexander, Channon
- Information Flow – Henry, Kafura

First are those based on **lexical content** such as Halstead, McCabe and Thayer measures. Halstead measures count total number of unique operators and operands. McCabe works with independent paths and Thayer introduces a measure that counts the occurrence of a wide variety of statement types.

Other measures concentrate on **information theoretic concepts**, such as entropy. These measures formulated through Alexander's work in architecture and design and Channon's work analyzing software structure. Unfortunately, these techniques require manual manipulation and are not proven as practical lexical measures.

A third type of measure deals directly with **information flow** through system connectivity. The importance of dealing with information flow is the ability to examine the software at design time, providing a quantitative assessment early. There is also the ability to automatically generate the measure versus a manual approach with measures that analyze the software structure. This premise holds specifically for Axiom 3.9.

F. SUMMARY

We find that in order to derive a measurement you have to understand the principles of measurement theory (i.e., the mapping of empirical relations to formal relations). You must understand the principles behind scale types (i.e., nominal, ordinal, interval, ratio and absolute); that there is some admissible transformation between the numbers in your scale. Finally, stated assumptions (i.e., axioms) are important when defining the measurement.

Measuring gives a quantitative result to an empirical understanding. It is easy to see one item bigger than another but that requires direct observation. Placing a value to that observation allows others the understanding of its relational composition. In time, and when standards of measurement have been met, the understanding becomes commonplace because a scale is built.

The basic rules of measurement apply also to software complexity. A scale and scale type need to be decided to provide meaning to the measurement. We can't say that 40 degrees is twice as warm as 20 degrees, but we can say that 40 feet is twice as long as 20 feet. The idea of temperature versus length is intuitively and fundamentally different.

This was not understood so well before standard practices upon which to measure temperature and length were decided and units of measurement placed.

In the beginning, there were multiple methods and units available to measure temperature and length. There was no standard. So, too, is the case with software complexity today. In time, there may be standards by which all software complexity is measured. In time, we will know just how complex a program measuring 50 is. But 50 what? The standard methods will help but they will not curb the different units of measurement. The standards can be chosen somewhat subjectively so long as the formal/quantitative relation continues to map to the empirical relation. These relations are the only manner in which to establish the standards. Axioms (i.e., empirical assumptions) must be explicitly stated to understand the mapping that occurs.

IV. PSDL CHARACTERISTICS

A. OVERVIEW

Appendix A [Ref. 2] provides a brief description of the CAPS process. CAPS performs rapid prototyping for real time systems by providing developers with a tool that maps visual graphics to a specification language (i.e., PSDL). PSDL is the key component to CAPS used at the design level to help flesh out requirements and can be incorporated into system feasibility studies. With the graphics editor, a skeleton of the system can be quickly drawn using an enhanced dataflow diagram with real time constraints.

Rapid prototyping is particularly effective for ensuring that the requirements accurately reflect the user's real needs, increasing reliability, reducing costly requirements changes [Ref. 6] and helps in estimating costs of the intended system. The process of rapid prototyping is an iterative one. The intent is not to build a fully executable system; it is to quickly build a working model that can be implemented with and by the user to ensure the needs of the user are being met. The prototype is not intended to be the final system. CAPS provides the capability of evolutionary prototyping, however, many times prototypes will not be included in the development of the actual system.

Essential in rapid prototyping is a working model that is easy to understand and modify. Software engineers will find themselves using the model while working with users, possibly making changes on the spot, as the problem and solution domains are better understood and defined. The most effective approach to rapid prototyping is through modularity. Modularity provides a quick approach with less coupling, while increasing understandability, reliability and maintainability of the actual system.

B. PSDL COMPUTATION MODEL

Luqi and Berzins [Ref. 6] make available an excellent description of executable PSDL.

PSDL is based on the enhanced dataflow diagram, a directed graph with associated timing and control constraints. The nodes/vertices of the graph are operators; the edges are data streams.

Operators are either functions (without an internal state) or state machines (with an internal state). When an operator fires, as a consumer, it reads one input value from each incoming edge; as a producer, it puts at most one computed output value on each outgoing edge. An operator's firing can be triggered by the arrival of a specified set of input data values, from a set of edges, or by a periodic timing constraint. There are two kinds of operators, atomic or composite. Atomic operators can be found in a software database or supplied by the software engineer. Composite operators abstract one or more operators. Fully decomposed, composite operators contain smaller dataflow diagrams of atomic operators.

The firing of an operator and the production of an output value can also be subject to conditional control constraints that depend on locally available data values. This limited facility for interconnecting operators is well matched to the needs of real-time systems, where each operator must complete its task in a fixed time.

A data stream carries values of an abstract or primitive data type. Both the built-in and user-definable PSDL data types are immutable. An immutable type has no operations for changing the state of a data object, so all changes appear as newly generated data values rather than as updates to existing data objects.

The generic built-in PSDL types include tuples (records), one-ofs (tagged variants), sets, sequences, maps (lookup tables), and relations. These types provide a powerful facility for defining finite collections of any value type and make it easy to construct many user-defined abstract data types. PSDL also has primitive data types for numbers, strings, and truth-values.

Each data stream is either a dataflow stream, which guarantees that each data element that enter is delivered exactly once, or a sampled stream, which guarantees that a data element can always be entered into or delivered from the stream on demand, at the cost of replicating elements or discarding older values. A dataflow stream acts like a first-in, first-out queue whose length is bounded by one. A sampled stream acts like a memory cell that always contains the most recent data value in the stream and that can be updated at any time.

In PSDL, the control and timing constraints of the operator receiving a stream determine whether the stream is a dataflow or sampled stream. If the triggering of an operator occurs only when all data arrives from a set of data streams, those data streams in the set are considered dataflow streams, otherwise they are considered sampled streams. Dataflow streams are discrete dataflows; sampled streams are continuous dataflows. The constraints guarantee there will be data values on all the input streams of an operator whenever it fires. Exceptions are treated as data values of a special data type, which flow down data streams subject to the same rules as ordinary data values.

Each operator can have a maximum execution time (MET) and a maximum response time (MRT), which are treated as hard real-time constraints. Operators with real-time constraints are periodic (synchronous) or sporadic (asynchronous). Giving its period specifies the firing frequency of each periodic operator. The minimum calling period (MCP) between firings is also specified for each sporadic operator to record the necessary assumptions about worst-case operation conditions for asynchronous external events.

You can also associate control constraints with operators. These include conditions that act as output guards for firing an operator, or passing an output value to a data stream, and as exception guards to control exception conditions or timers.

It is easy to describe individual timing constraints of a real-time system, but large real-time systems often contain a mixture of periodic and sporadic operators with many different frequencies. The interactions between such timing constraints can be very complex and very difficult to analyze without the help of a computer.

C. PSDL COMPLEXITY

In an empirical sense, we find the complexity of PSDL to be based on the following characteristics and the properties of operators and data streams:

- Degree of modularity/decomposition

- Number and type of operators/nodes (atomic, composite, external systems or internal software)
- Number and type of data streams/edges (dataflow, sampled or state streams)
- Timing constraints (MET, MRT, MCP)
- Control constraints (triggering, guards, periodic or sporadic firing)
- Number of unique types (abstract versus primitive data types)
- Degree of understandability, maintainability, and reliability of the final system.

Many complexity measures look first to the code and then represent that code as a dataflow diagram (e. g., McCabe's Complexity measures). We are at an advantage with PSDL because it is auto-generated code from a CAPS dataflow diagram. Only in empirical terms, do we need not concern ourselves with the actual PSDL code; the diagram itself more easily represents its complexity (holds for Axioms 3.5 and 3.10). Subsequently, in the simplest of terms, to decrease complexity of PSDL we need to increase the understandability of the dataflow diagram. Although PSDL can support evolutionary prototyping, we will not concern ourselves with its ability to be maintained or to be completely reliable when considering complexity. Because many prototypes are of a throwaway nature, not intended to evolve into actual systems, we will concentrate solely on complexity in terms of understandability (i.e., Axiom 3.4). The scope of complexity measurements, itself, should remain simple (i.e., Axiom 3.9, 3.10). This approach does not remove the importance of or correlation between complexity, maintainability and reliability. In the words of McCabe [Ref. 7]:

> Overly complex modules are more prone to error, are harder to understand, are harder to test, and are harder to modify. Deliberately limiting complexity at all stages of software development, for example as a departmental standard, helps avoid the pitfalls associated with high complexity software.

Therefore, reducing complexity through understandability will implicitly increase maintainability and reliability. There are direct relationships between them all.

Initially, the dataflow diagram represents a subset of system requirements that the software engineer chose to implement, to aid in the requirements process, and to determine if real time constraints of those features can be satisfied. Essentially the prototype tries to determine if the system will behave according to its specification. A software engineer can use the prototype and representative dataflow diagram to work side by side with the user and explain their understanding of the problem domain.

Concurrently, they will be explaining the solution domain to the user. It is imperative that the diagram is easily understood to facilitate this process. Later, during the design process, the software engineer will include more features in the prototype to represent the proposed system more accurately and completely. This prototype, if elected, can go through an evolutionary approach for development of the actual system. Again, it is very important to alleviate the complexity of the diagram to increase understandability by other software engineers and system engineers who may take part in the development of the intended system.

Figure 4.1 and 4.2 represent two separate dataflow diagrams from CAPS to illustrate the empirical meaning behind understandability and complexity. The figures illustrate the difference in complexity. Figure 4.1 represents the dataflow diagram for an Autopilot Control System and Figure 4.2 for a Fish Farm Control System. The sheer difference in numbers of nodes and edges (i.e., 12 versus 26 respectively) could represent greater complexity in Figure 4.2 versus 4.1. Empirically and in terms of understandability, Figure 4.2 certainly seems more complex than Figure 4.1.

Should like systems represented by two different diagrams with an unequal number of nodes and edges be considered equally complex? Should distinct systems, represented by diagrams of equal nodes and edges, be considered equally complex? Figure 4.3 represents another Fish Farm Control System. The total number of nodes and edges for Figure 4.2 is 26 and for Figure 4.3 is 24. They are like systems, performing the same function, and have unequal complexities. Thus, it is quite possible to have like systems of different complexity based on the software engineer's interpretation of the requirements. However, relying solely on counting nodes and edges merely represents a measure of size versus complexity and may not be the best representation, as described in Chapter I and II and Axiom 3.7.

Axiom 3.11 also told us that there is more to a program's complexity than the functions it performs. Where we have like systems with unequal complexities, we can also have distinct systems, of equal nodes and edges, with unequal complexities. It is quite obvious at this point that in order to identify the complexity properly, we cannot look only to the functions a program performs (holds for Axiom 3.11) or to the size of the program (holds for Axiom 3.7). You must turn to other properties to measure complexity accurately.

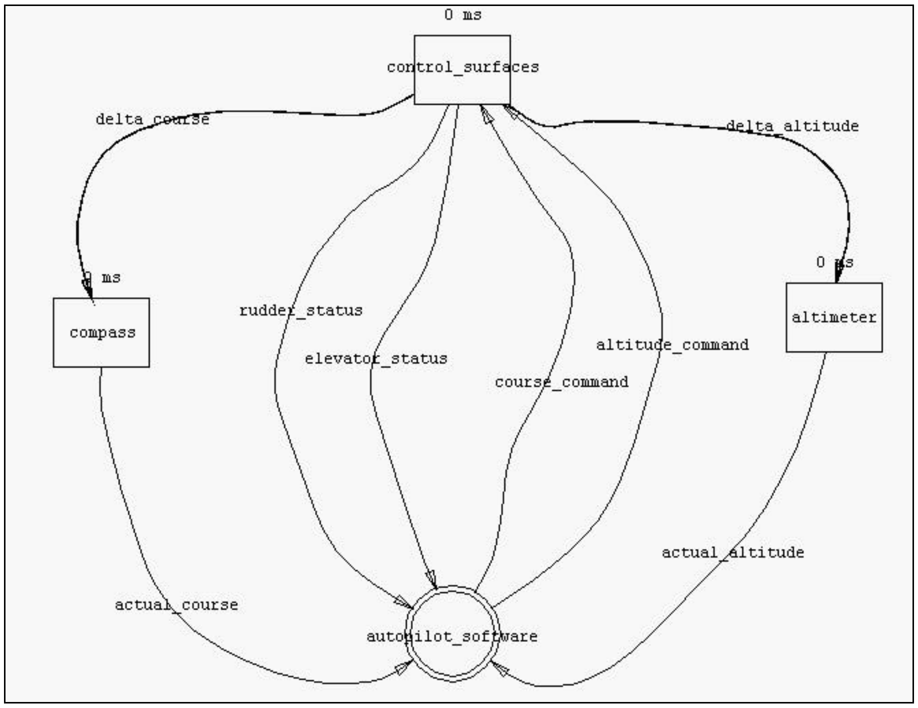


Figure 4.1. CAPS Dataflow Diagram of an Autopilot Control System.

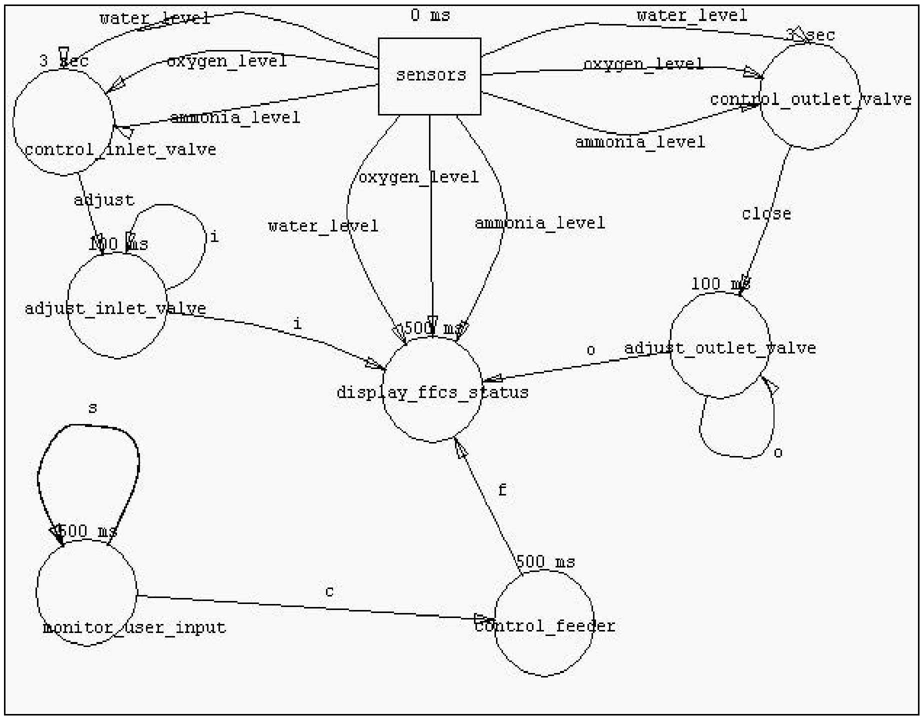


Figure 4.2. CAPS Dataflow Diagram of Fish Farm Control System I.

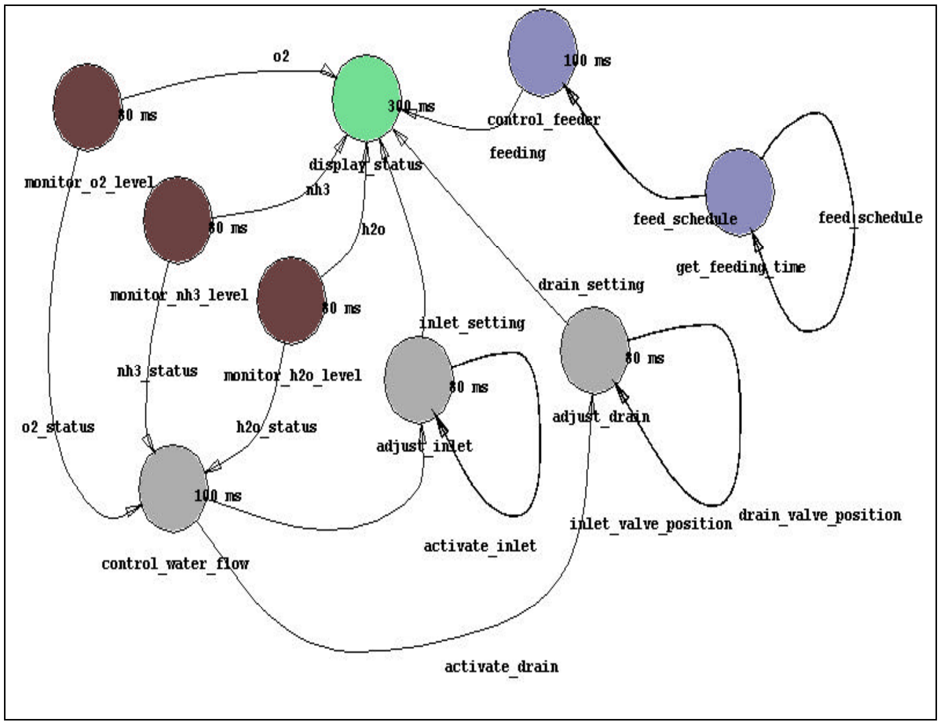


Figure 4.3. CAPS Dataflow Diagram of Fish Farm Control System II.

D. DECOMPOSITION AS A CHARACTERISTIC

From a systems engineering standpoint, we decompose complex systems to the lowest level needed to increase our understanding of the system. CAPS approaches its design in a similar top-down fashion for much the same reason. When building the model we try to keep the numbers of nodes in the drawing pane to 7 ± 2 for simplicity and ease of understanding. This notion fits into the theory that human short term or working memory can only process 5-9 "chunks" of information where a chunk is any meaningful unit of measurement. [Ref. 9] In this case, a chunk could be considered a single node in the diagram. To assist with this notion CAPS makes use of composite operators. Abstracted under a composite operator is the substructure, another dataflow diagram, to complete the system. If increasing the modularity through composite operators in the requisite diagram increases understandability, it should decrease the overall complexity. From another perspective, decomposition of modules helps to refine or hone the prototype by increasing the reuse of existing components. Reuse is another method of keeping complexity in check.

Reusing the Autopilot Control System of Figure 4.1 we can see the composite operator, *autopilot_software* being used, represented as nested nodes. Figure 4.4 shows the decomposition of the operator.

To accurately represent complexity of the entire Autopilot Control System we need to take into consideration the decomposition of *autopilot_software*, (Figure 4.4). From a strict additive sense, Figure 4.1's 12 nodes and edges combined with Figure 4.4's 11 nodes and edges (note: edges *actual_altitude* and *actual_course* were only counted once) give us a complexity of 23 for the entire system. By Axiom 3.13, if we treat the diagram as a composition of the top-level structure and the substructure, without composite operators and modularity, it should be more complex than the sum of each individual diagram.

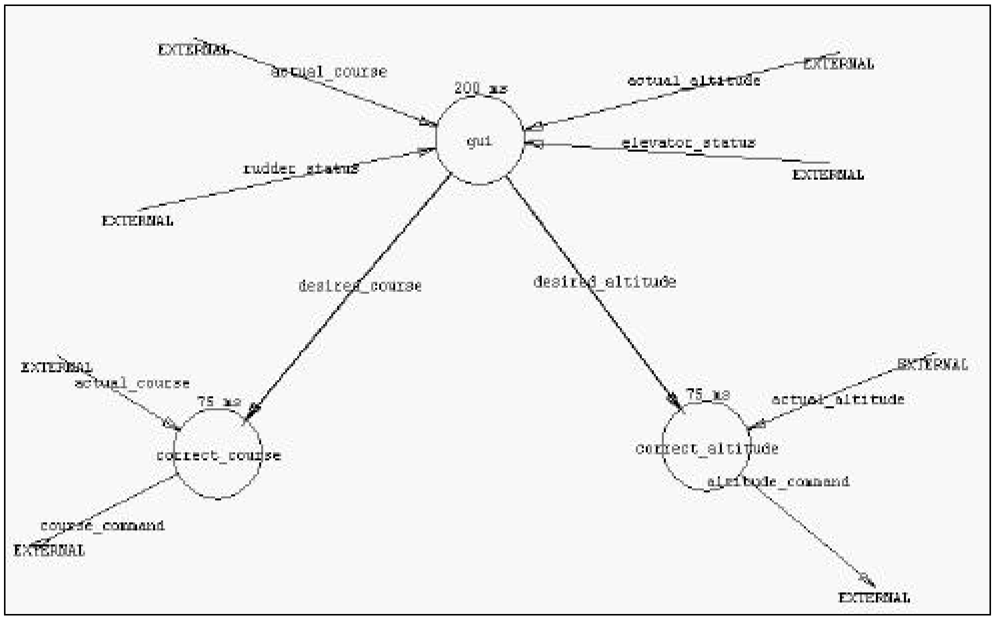


Figure 4.4. Decomposition of *autopilot_software.*

Expanding Figure 4.1 by concatenating Figure 4.4 yields a flattened diagram represented in Figure 4.5. Here we have a total of 16 nodes and edges versus the sum of 23 (note: *actual_altitude* and *actual_course* are represented in this diagram as a hyper-edge but only counted once). Clearly, expanding the diagram presents us with a representation using no composite operators, and no modularity, (i.e., a more complex diagram), but the complexity value does not represent this (does not hold for Axiom 3.5). We wish to say that building our system with composite operators has a positive affect on understandability by decreasing the complexity. Simply viewing complexity by expanding the diagram does not present us with this fact. We could have simply built the model without composite operators in the first place.

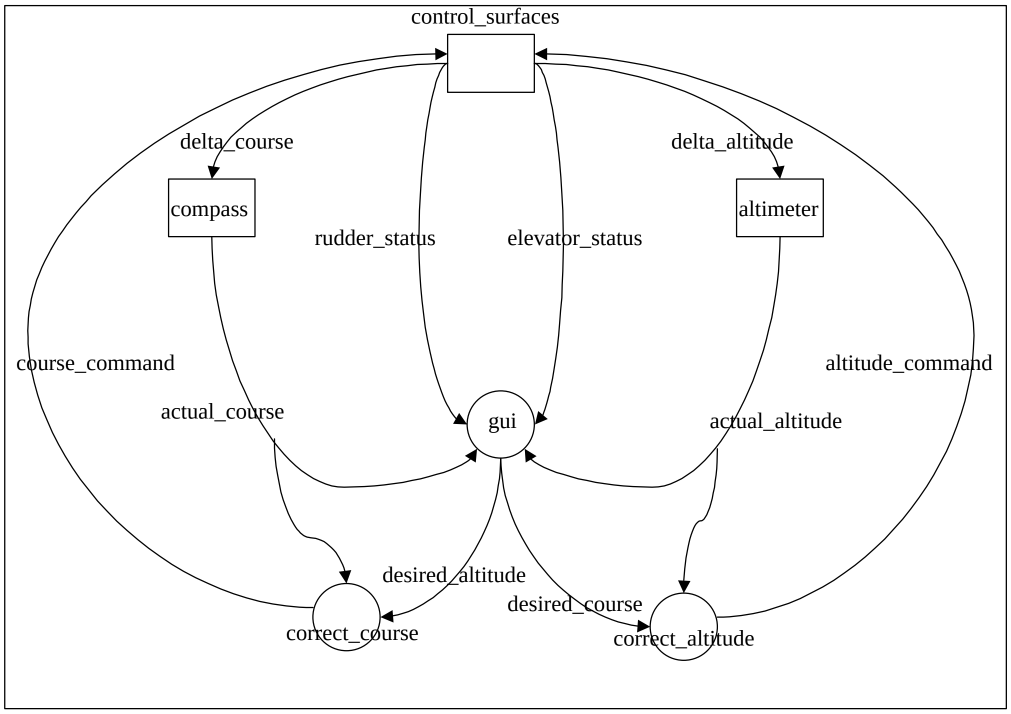


Figure 4.5. Expanded Autopilot Control System.

Another possibility is to view the diagrams separately, as in the strict additive case, but instead of counting each edge of Figure 4.4, count only those edges not represented in the original diagram of Figure 4.1. In Figure 4.4 we see those edges without any EXTERNAL markings and labeled *desired_course* and *desired_altitude.* This approach also holds for Axiom 3.6 where if we counted each node and edge, of Figure 4.4, we would have 13, which is unacceptable.

Figure 4.4, then, gives us a possible 5 nodes and operators that can then be added to the original 12 for a complexity of 17. Unfortunately, 17 represents greater complexity than our flattened diagram of 16. For Axiom3.5 and 3.13 to hold, we need to find a way to quantify the complexity where the measurement falls between a minimum and maximum (i.e., 12 and 16 in this example).

E. SUMMARY

PSDL is auto-generated specification code from CAPS. CAPS was developed as a tool for rapid prototyping of real-time systems. When building a prototype in CAPS, it is essential the working model is easy to understand and modify (holds for Axiom 3.9 and 3.10). This is partly accomplished by CAPS' ability to translate enhanced dataflow diagrams into PSDL. The PSDL computational model is represented as an augmented graph: $G = (V,E,T(v),C(v))$, where:

- V is a set of vertices (v) representing operators.
- E is a set of edges representing data streams.
- T(v) is the set of timing constraints for each node *v*.
- C(v) is the set of control constraints for each node *v*.

Each operator and data stream can have associated properties that affect the production, consumption and flow of data. For operators the properties are timing and control constraints; for data streams the properties are related to latency timing. We also find that operators can be represented as atomic or composite operators, and data streams can be represented as dataflow or sampled streams. The behavior of each operator and data stream depends on its properties and type. Each type of operator and data stream has its own complexity, which changes with additional properties.

To fully investigate complexity of PSDL we must, therefore, take into consideration the type of and properties associated with operators and data streams. Part of this is to properly account for composite operators that are abstracted substructures.

THIS PAGE INTENTIONALLY LEFT BLANK

V. PSDL COMPLEXITY MEASURE

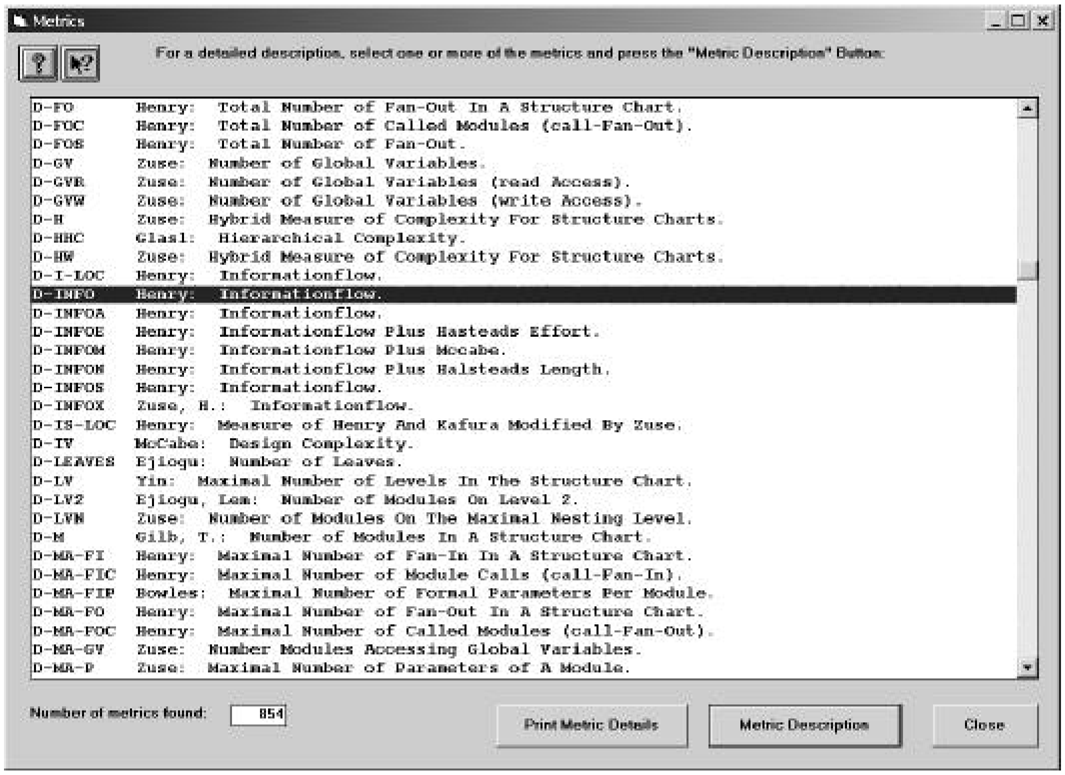
A. OVERVIEW

A proper measurement is one that will answer the question *"What do I want to learn?"* or *"What is the measurement goal?"* These questions help to identify the purpose of the measurement. A complexity measurement should represent our intuitive idea of complexity. [Ref. 14]

For PSDL we need to *learn* about its characteristics in order to recognize the attributes that contribute to its complexity. In Chapter IV, we defined its complexity to be the <u>understandability</u> of the dataflow diagram defined by $G = (V, E, T(v), C(v))$ (see also Appendix A [Ref. 2]). Our intuition tells us a diagram with many nodes, edges and constraints is one of great complexity; the *goal* then is to model systems using as few of these attributes as possible. Furthermore, we stated that the use of composite operators gives us a diagram (i.e., a system) with greater understandability, hence, less complexity.

To accomplish this goal we could have started with a clean slate, developing a complexity measure that is unique to PSDL. Instead, we find that Zuse [Ref. 14, 15] provides us with a list of nearly 1500 software measures, 100 of which deal with complexity directly. We find those measures available in a broad software application called Zuse/Drabe Measure Information System (ZD-MIS) [Ref. 16] that contains an extensive database. Figure 5.1 is a screen shot of this application showing an alphabetical listing of a subset of the measures, A-M. There are 854 measures, A-M. The measure *D-INFO* is highlighted because it will be discussed later in this chapter.

To make use of these measures we need to consider and study their properties to determine any possible correlation to PSDL. We discussed in Chapters I, II and III how sizing measures may or may not be proper to represent complexity. We showed in Chapter IV how one additive sizing measure did not hold for Axiom 3.13. Therefore, sizing measures are not considered because they do not represent our intuitive idea of complexity for dataflow diagrams.


Metrics
For a detailed description, select one or more of the metrics and press the "Metric Description" Button:

```
D-FO       Henry:  Total Number of Fan-Out In A Structure Chart.
D-FOC      Henry:  Total Number of Called Modules (call-Fan-Out).
D-FOS      Henry:  Total Number of Fan-Out.
D-GV       Zuse:  Number of Global Variables.
D-GVR      Zuse:  Number of Global Variables (read Access).
D-GVW      Zuse:  Number of Global Variables (write Access).
D-H        Zuse:  Hybrid Measure of Complexity For Structure Charts.
D-HHC      Glasl:  Hierarchical Complexity.
D-HW       Zuse:  Hybrid Measure of Complexity For Structure Charts.
D-I-LOC    Henry:  Informationflow.
D-INFO     Henry:  Informationflow.
D-INFOA    Henry:  Informationflow.
D-INFOE    Henry:  Informationflow Plus Hasteads Effort.
D-INFOM    Henry:  Informationflow Plus Mccabe.
D-INFON    Henry:  Informationflow Plus Halsteads Length.
D-INFOS    Henry:  Informationflow.
D-INFOX    Zuse, H.:  Informationflow.
D-IS-LOC   Henry:  Measure of Henry And Kafura Modified By Zuse.
D-IV       McCabe:  Design Complexity.
D-LEAVES   Ejiogu:  Number of Leaves.
D-LV       Yin:  Maximal Number of Levels In The Structure Chart.
D-LV2      Ejiogu, Lem:  Number of Modules On Level 2.
D-LVN      Zuse:  Number of Modules On The Maximal Nesting Level.
D-M        Gilb, T.:  Number of Modules In A Structure Chart.
D-MA-FI    Henry:  Maximal Number of Fan-In In A Structure Chart.
D-MA-FIC   Henry:  Maximal Number of Module Calls (call-Fan-In).
D-MA-FIP   Bowles:  Maximal Number of Formal Parameters Per Module.
D-MA-FO    Henry:  Maximal Number of Fan-Out In A Structure Chart.
D-MA-FOC   Henry:  Maximal Number of Called Modules (call-Fan-Out).
D-MA-GV    Zuse:  Number Modules Accessing Global Variables.
D-MA-P     Zuse:  Maximal Number of Parameters of A Module.
```

Number of metrics found: 854

Figure 5.1. ZD-MIS Database. "From [Ref. 16]"

B. EARLY CONSIDERATIONS

Noguiera [Ref. 10] presented two PSDL complexity measures: Large Granular Complexity (**LGC**) and Fine Granular Complexity (**FGC**). **LGC** was mentioned in Chapter II and is presented in detail in Appendix A. [Ref. 2] LGC is not being considered because it represents a similar notion of strictly adding the number of nodes and edges together. In Chapter IV we presented the same notion and quickly established it had no validity under the premises of understandability and did not hold for Axiom 3.13.

His second, **FGC**, is taken literally as fan_in + fan_out of data streams into and out of each operator:

Equation 5.1: $FGC(o_i) = fan\_in + fan\_out$

where:

o_i is a particular operator

This measure is presented as the relational complexity of the number of data streams into and out of an operator and when that number tends to get large, the designer should consider the use of composite operators to reduce complexity.

Measures based on information flow also propose the idea of fan_in and fan_out (i.e., information going into and out of modules). From Zuse [Ref. 15, 16] we learn many things about these types of measures that can then be used to evaluate their applicability to PSDL, using the axioms of Chapter III. Information flow measures:

- lie on ordinal and ratio scales – Axiom 3.3 holds.
- are related to structure charts – Axiom 3.7 holds.
- can be used during the design, coding and testing phase – Axiom 3.8 holds.
- are related to entire systems and independent modules – Axioms 3.9-3.11 hold.
- are type classified as complexity and comprehendability (i.e., understandability) measures.

There are many information flow measures available but Henry and Kafura [Ref. 4] started the concept in the early 80’s with the following equation defined as a function of each module i:

Equation 5.2: $D - INFO = \left(\sum_{i=1}^{n} fi(i) * fo(i) \right)^2$

where:

- *D-INFO* is the name given to the measure by Zuse [Ref. 15]; it is a hyphenated abbreviation; the meaning of *D* could not be found, *INFO* is information.
- fi is fan_in of module i and $fi = \max(fan\_in, 1)$,
- fo is fan_out of module i and $fo = \max(fan\_out, 1)$,
- *n* is the total number of modules.

This complexity measure takes the product of the total number of fan_in and fan_out streams per module, representing the total possible number of combinations of fan_in streams to fan_out streams for the module. The complexity is then found by taking the sum of all possible number combinations for the system. The power factor of two is used based on earlier work in the laws of programmer interaction and system partitioning. [Ref. 4]

Zuse redefines the measure by dropping the square because it can be argued there is not much gained in an empirical sense by squaring the value so long as the value remains positive. Equation 5.3 will become the basis for the PSDL complexity measure.

Equation 5.3: $D - INFO' = \sum_{i=1}^{n} \left(fi(i) * fo(i) \right)$

where:

- *D-INFO*' is the name given to the refined measure.
- fi is fan_in of module i and $fi = \max\left(fan\_in, 1 \right)$,
- fo is fan_out of module i and $fo = \max\left(fan\_out, 1 \right)$,
- *n* is the total number of modules.

C. DEVELOPING THE COMPLEXITY MEASURE FOR PSDL

1. Complexity and Hybrid Measures

A complexity measure for PSDL needs to consider three things: **scale**, **properties**, and **information flow.** First, in Chapter III we defined **scales** and presented Axiom 3.3 that stated complexity measures must fall under ordinal or ratio scales. Next, in Chapter IV we stated the operators and data streams (i.e., nodes and edges) each have **properties** contributing to different levels of complexity. Finally, as depicted above, **information flow** seems to show some promise in determining the complexity of the dataflow diagram. Therefore, what we need is a hybrid measure that takes all of these into consideration.

When defining or developing hybrid measures, especially when combining two or more individual measures, it is important to ensure there is nothing lost, empirically or qualitatively, by combining those measures. It is also important that if measures are combined additively, that the units are accurately accounted for. Zuse [Ref. 15] provides and excellent example of this with the following:

Equation 5.4: $MCC - V = |E| - |N| + 2|M|$

Equation 5.5: $HC = \frac{|M|}{L}$

Equation 5.6: $SC = \frac{|A|}{|M|}$

Equation 5.7: $C = \frac{M^2 + AL + LM\left(E - N + 2M\right)}{LM}$

The individual complexity measures in Equation 5.4 (a sizing measure of McCabe), Equation 5.5 (a hierarchical complexity measure) and Equation 5.6 (a structural complexity measure) are combined as a sum to represent the whole system in Equation 5.7. Providing detail of these measures is not important to show the purpose of this example. It is clear that the hybrid measure in Equation 5.7 is not a good representation of the whole system because *M* modules and *L* levels in the hierarchy are represented in both the numerator and denominator. As *M* or *L* increase or decrease, *C* will not change proportionally, as we should expect empirically.

2. Complexity and Information Flow

Initial analysis considered Equation 5.8 as a possible complexity measure. It is similar to Nogueira's *FGC* in Equation 5.1. However, where *FGC* calculates a complexity of each individual operator, Equation 5.8 calculates complexity of the entire system by summing the individual operator complexities.

Equation 5.8: $C = \sum_{i=1}^{n} fan\_in\left(o_i\right) + fan\_out\left(o_i\right)$

where:

- *C* is the <u>total</u> complexity,
- *i* is an index for each operator,
- *n* is the total number of operators.

Equation 5.8 presents two key problems. First, using an additive value of fan_in and fan_out provides **an inaccurate representation** of input and output data streams. Second, it promotes **no intuitive understanding** of complexity. The best relationship for complexity is a multiplicative one as in Equation 5.3. Figure 5.2 shows an example of the additive versus multiplicative relationship of fan_in and fan_out.

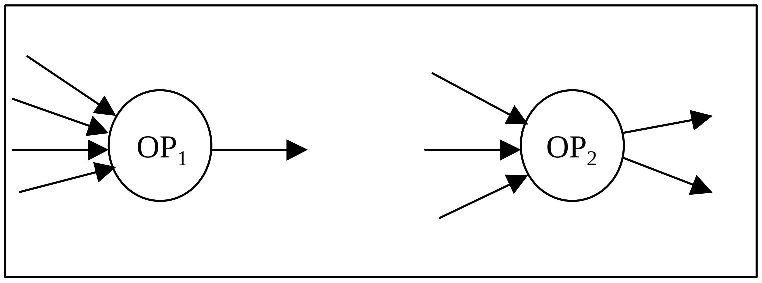


Figure 5.2. Relationship of Fan_in and Fan_out.

The dataflow diagram in Figure 5.2 is described by the tuple *G* = (*DSI*, *DSO*), where: *DSI* is data streams in, *DSO* is data streams out. The sum of data streams for OP_1 = (4, 1) and OP_2 = (3, 2) would be: 4 + 1 = 5 and 3 + 2 = 5. Whereas the product of those data streams is: 4 * 1 = 4 and 3 * 2 = 6. Certainly, the complexity of these two diagrams is not equal as is shown in the additive case. Empirically the complexity associated with OP_2 is greater than that of OP_1 as is shown in the multiplicative case. Additionally, the product of these streams represents all possible combinations of input data streams to output data streams. As these combinations increase, so too, should the complexity.

3. Complexity and PSDL Properties

Information flow and the product of input and output data streams provide a good starting point to measure complexity of PSDL. Further, the properties of the data streams and operators require consideration. In Chapter IV we briefly defined some characteristics of PSDL and touched upon some of the properties associated with operators and data streams. Table 5.1 lists those properties in more detail, presenting them on an ordinal scale from greatest to least complexity.

Table 5.1 shows internal software components more complex than external systems. Additionally, composite operators are more complex than atomic operators. It also shows abstract data types being more complex than primitive types and dataflow streams of more complexity than sampled streams.

OPERATORS	**DATA STREAMS**
A. Internal Software Components (ISC)	A. Abstract Data Types (ADT)
1. Composite Operators (CO)	1. Dataflow Streams (DF)
a. Timing Constraints	a. Latency
1.) Max Execution Time (MET)	b. No Latency
2.) Sporadic Timing	2. Sampled Streams
a.) Max Response Time (MRT)	a. State Stream (SS)
b.) Min Calling Period (MCP)	1.) Latency (L)
3.) Periodic Timing	2.) No Latency (NL)
a.) Period (P)	b. Non-State Stream (NSS)
b.) Deadline (D)	1.) Latency (L)
b. Control Constraints	2.) No Latency (NL)
1.) Trigger by All (TA)	B. Primitive Types (PT)
2.) Trigger by Some (TS)	1. Dataflow Streams (DF)
3.) Execution Guards (EG)	a. Latency (L)
2. Atomic Operators (AO)	b. No Latency (NL)
a. Timing Constraints ⋮	2. Sampled Streams
b. Control Constraints ⋮	a. State Stream (SS)
B. External Systems (ES)	1.) Latency (L)
1. Composite Operators (CO) ⋮	2.) No Latency (NL)
2. Atomic Operators (AO) ⋮	b. Non-State Stream (NSS)
	1.) Latency (L)
	2.) No Latency (NL)

Table 5.1. PSDL Properties.

Seen directly in a PSDL dataflow diagram (refer to Figure 4.1) are several of the properties listed in Table 5.1:

- Internal Software Components (ISC) – circular nodes.
- External Systems (ES) – rectangular nodes.
- MET properties – labeled directly above the node.
- Composite Operators (CO) – nested circular nodes.
- Data streams – arrows
- State Streams (SS) – bold arrows
- Instantiations of data types – stream labels.

The ordinal scale of Table 5.1 is partly subjective, partly objective. **Internal software components** are internal to the system being modeled whereas **external systems** can be thought of as black boxes affecting the modeled system, most often as sensors. **Internal software components** are controlled by the designer and represent greater complexity over the black box **external systems**. **Composite operators** have underlying dataflow diagrams and are clearly more complex than **atomic operators**, which are at their lowest level. During program execution, **timing constraints** are evaluated prior to **control constraints** representing greater affect, hence greater complexity. The individual properties associated with timing and control constraints are also evaluated in a particular order, which is taken to represent their complexity also. **Periodic operators** have regular schedules, are more deterministic, and therefore represent less complexity than its counterpart **sporadic operators**.

In the data stream column, **abstract data types** are represented as being more complex than **primitive types** because of the inherent complexity associated with the two types. **Dataflow streams** are discreet providing one value to an operator at a time whereas **sampled streams** are continuous, guaranteeing delivery of data on demand. This availability of data is interpreted as dataflow streams being more complex than sampled streams. **State streams** represent a property of sampled streams. A state stream has greater affect on the system and is considered more complex. Lastly, any data stream that has **latency** timing associated with it has additional properties and is considered more complex.

How to use this hierarchical nature of PSDL properties seemed intuitive at this point. By weighting each property and adding that weight to each individual operator and data stream, you could represent the value of each operator and data stream as some value greater than itself. Refer back to Figure 5.2 for the following explanation.

By taking the product of *DSI* and *DSO* we represented all possible combinations of data streams in to data streams out. In that particular example: OP_1 = (4, 1) and OP_2 = (3, 2), 4 * 1 = 4 and 3 * 2 = 6. By adding a weighted value to just the data streams, in this example, OP_1 = (some number $a > 4$, some number $b > 1$) and OP_2 = (some number $x > 3$, some number $y > 2$). Taking the product now yields: a * b > 4 and x * y > 6. What this tells us is that the complexity is no longer a simple product of fan_in and fan_out, it is something greater than that. How much greater is dependent upon the weighting factor i.e., the PSDL properties associated with the data streams. This does not change the intuitive understanding of complexity. In fact, it presents it more accurately by considering the properties associated with PSDL. The idea of weighting factors also holds for Axioms 3.3, 3.5, 3.8 and 3.9.

The example above showed the affect of applying a weighting factor to data streams. More importantly it showed the validity of using weighting factors to determine complexity. It was provided to lead us into the next step of building the complexity measure. That step is the application of weighting to the operators. The base Equation 5.3 is used:

$$C = D - INFO' = \sum_{i=1}^{n} \left(fi(i) * fo(i) \right)$$

Equation 5.3 actually represents fan_in and fan_out per operator and can be represented as:

Equation 5.9: $C = D - INFO' = \sum_{i=1}^{n} \left(\frac{fi(o_i) * fo(o_i)}{o_i} \right)$

where:

- C is complexity,
- *D-INFO*$'$ is the name given to Zuse's [Ref. 15] refined measure,

- fi is fan_in of operator *o_i* and $fi = \max\left(fan\_in, 1 \right)$,
- fo is fan_out of operator o_i and $fo = \max\left(fan\_out, 1 \right)$,
- $o_i = 1 + w_i$,
- w_i is some weighting factor applied to each operator based on its individual properties.
- *n* is the total number of modules.

Changing some variables from Equation 5.9, to use PSDL terminology, the equation can be represented as:

Equation 5.10: $DS = \sum_{i=1}^{n} \frac{dsi\left(o_i\right) * dso\left(o_i\right)}{o_i}$

where:

- *DS* is complexity on the Dupont Scale,
- *dsi* is data streams in of operator o_i and *dsi* = max(*data_stream_in*, 1),
- *dso* is data steams out of operator o_i and *dso* = max(*data_stream_out*, 1),
- o_i is each individual operator and $o_i = 1 + w_i$,
- *n* is the total number of operators.

4. Complexity and PSDL Weighting Factors

The inclusion of weighting factors to the data streams and operators under a base equation offers a hybrid complexity measure for PSDL that now includes information flow and, PSDL properties and presents them on a ratio scale. However, adding weighting factors to each operator as in, $o_i = 1 + w_i$, generates a number in the denominator greater than one. This means that as the complexity of each operator increases, the complexity of the system (i.e., the model), will decrease. This obviously is not intuitive and makes the hybrid measure invalid. Instead we shall represent the complexity equation as:

Equation 5.11: $DS = \sum_{i=1}^{n} \frac{dsi\left(o_i\right) * dso\left(o_i\right)}{o_i^{-1}}$ or:

Equation 5.12: $DS = \sum_{i=1}^{n} o_i \left[dsi\left(o_i\right) * dso\left(o_i\right) \right]$

As more weighting is applied to increase the complexity of each data stream and operator, the complexity of the system will also increase.

5. The Weighting Tables

Defining the weights was the next step to defining the measure. This was accomplished by building a table that included all possible combinations of properties inherent to data streams and operators. Using Table 5.1, as a basis, the weighting tables were built in order of greatest to least complexity, left to right and top to bottom. The x-axis represents all possible **types** of operators and data streams and the y-axis represents all possible timing and/or control **constraints**. Table 5.2 is a table of weighting factors for operators and Table 5.3 is a table of weighting factors for data streams. Important to note is the mutually exclusive relationships that exist.

- All types of operators and data streams are mutually exclusive.
- MRT/MCP and Period/Deadline are mutually exclusive.
- TA and TS are mutually exclusive.
- L and NL are obviously mutually exclusive.

Type *(w(t))* / **Constraint** *(w(c))*	ISC & CO (0.4)	ISC & AO (0.3)	ES & CO (0.2)	ES & AO (0.1)
MET (0.200)	0.080	0.060	0.040	0.020
MRT (0.178)	0.071	0.053	0.036	0.018
MCP (0.156)	0.062	0.047	0.031	0.016
Period (P) (0.133)	0.053	0.040	0.027	0.013
Deadline (D) (0.111)	0.044	0.033	0.022	0.011
Trigger by All (TA) (0.089)	0.034	0.027	0.018	0.009
Trigger by Some (TS) (0.067)	0.027	0.020	0.013	0.007
Execution Guards(EG)(0.044)	0.018	0.013	0.009	0.004
None (N) (0.022)	0.009	0.007	0.004	0.002

Table 5.2. Weighting for **Operators,** (w).

Type *(w(t))* / **Constraint** *(w(c))*	ADT & DF (0.286)	ADT & SS (0.238)	ADT & NSS (0.190)	PT & DF (0.143)	PT & SS (0.095)	PT & NSS (0.048)
Latency (L) (0.667)	0.191	0.159	0.127	0.095	0.063	0.032
No Latency (NL) (0.333)	0.095	0.079	0.063	0.048	0.032	0.016

Table 5.3. Weighting for **Data Streams,** (e).

The individual weighting factors (i.e., *w(t)* and *w(c)*) for each type and constraint in Tables 5.2 and 5.3 were determined as follows:

- Each type and constraint was assigned a rank 1 – *n* based on its complexity from the hierarchy in Table 5.1. The least complex received a 1; the most complex received a value of *n.*
- The associated weight of each type and constraint was equal to its rank divided by the sum of ranks:

Equation 5.13.1: $w(t_i) = i \Big/ \sum_{i=1}^{n} i$

Equation 5.13.2: $\sum_{i=1}^{n} w(t_i) = 1$.

Table 5.4 provides an example.

TYPE (*t*)	**RANK** (*i*)	**WEIGHT** ($w(t_i)$)
ISC & CO	1	.1
ISC & AO	2	.2
ES & CO	3	.3
ES & AO	4	.4

Table 5.4. Individual Weighting Factors.

The weighting factors in the cells of each table were derived by using the cross product of *w(t)* and *w(c).*

6. PSDL Complexity Measure

Reusing the complexity measure given as Equation 5.12 and adding what we now know about weighting factors, the complexity measure for PSDL can be fully described and instructions provided for its use.

$$DS = \sum_{i=1}^{n} o_i \left[dsi\left(o_i\right) * dso\left(o_i\right) \right]$$

where:

- DS is complexity of PSDL under the Dupont Scale,
- o_i is each individual operator,
- *dsi* is data streams in of operator o_i and *dsi* = max(*data_stream_in*, 1),
- *dso* is data steams out of operator o_i and *dso* = max(*data_stream_out*, 1),
- *n* is the total number of operators.

$$\begin{aligned} o_i &= 1 + w_i \\ &= 1 + \sum_{\substack{1 \le x \le 4 \\ 1 \le y \le 9}} w\left[x, y\right] * U_i\left[x, y\right] \end{aligned}$$

where: $U_i\left[x, y\right] = 1$ if o_i is of type x and has constraint y,

$U_i\left[x, y\right] = 0$ otherwise.

$$\begin{aligned} dsi\left(o_i\right) &= \sum_{j=1}^{e_{in}} ds_j \\ &= \sum_{j=1}^{e_{in}} 1 + e_j \\ &= \sum_{j=1}^{e_{in}} \left(1 + \sum_{\substack{1 \le x \le 6 \\ 1 \le y \le 2}} e\left[x, y\right] * U_j\left[x, y\right] \right) \end{aligned}$$

where: $U_j\left[x, y\right] = 1$ if ds_j is a data stream in and is of type x and has constraint y,

$U_j\left[x, y\right] = 0$ otherwise,

e_{in} is the total number of data streams in for operator o_i.

$$dso(o_i) = \sum_{k=1}^{e_{out}} ds_k$$

$$= \sum_{k=1}^{e_{out}} 1 + e_k$$

$$= \sum_{k=1}^{e_{out}} \left(1 + \sum_{\substack{1 \le x \le 4 \\ 1 \le y \le 2}} e[x,y] * U_k[x,y] \right)$$

where: $U_k[x,y] = 1$ if ds_j is a data stream out and is of type x and has constraint y,

$U_k[x,y] = 0$ otherwise,

e_{out} is the total number of data streams out for operator o_i.

D. AN EXAMPLE OF THE COMPLEXITY MEASURE FOR PSDL

To demonstrate how to calculate complexity of PSDL models, we will use the Autopilot Control System of Chapter IV. Some information needed to calculate its complexity can be found directly from the augmented graph of Figure 4.1 and Figure 4.4. For instance, you can easily determine the number of operators and the number of input/output data streams for each operator. Bold arrows represent state streams, MET values are listed above each node, and nested nodes easily identify composite operators. Unfortunately, you must go directly to the PSDL code to find the remaining timing and control constraints.

CAPS generates two *.PSDL files for every version of every prototype: an **expanded** file and a **source** file. The **source** file can be found in the “version” directory, which is located under the “root” directory. Root directory names are equivalent to the source file name. The source file is the code generated by the original system diagram drawn by the user (See Figure 4.1). It contains composite operators, if any. The PSDL source file for the Autopilot Control System can be found in Appendix C. The **expanded** file is located in a <<Temp>> subdirectory of the “version” directory. This file is code generated by CAPS, representing the system as a flattened hierarchy without composite operators (see Figure 4.5). The PSDL expanded file for the Autopilot Control System can be found in Appendix A [Ref. 2] as well as other examples.

The information for Table 5.5 was extracted from the PSDL source file for Autopilot Control System (see Appendix C). That information is highlighted to show where it can be found. This table shows each operator and data stream with their respective weights. It also shows the individual properties of each operator and data stream along with their respective weights.

OPERATOR		**DATA STREAM**	
o_1 – control_surfaces	(1.061)	ds_1 – delta_course	(1.032)
• ES & AO		•	PT & SS
–	MET (0.020)	–	NL (0.032)
–	MCP (0.016)		
–	MRT (0.018)		
–	TS (0.007)		
o_2 – compass	(1.033)	ds_2 – delta_altitude	(1.032)
• ES & AO		•	PT & SS
–	MET (0.020)	–	NL (0.032)
–	P (0.013)		
o_3 – altimeter	(1.033)	ds_3 – rudder_status	(1.063)
• ES & AO		•	ADT & NSS
–	MET (0.020)	–	NL (0.063)
–	P (0.013)		
o_4 – autopilot_software	(1.009)	ds_4 – elevator_status	(1.063)
• ISC & CO		•	ADT & NSS
		–	NL (0.063)
o_5 – gui	(1.113)	ds_5 – course_command	(1.063)
• ISC & AO		•	ADT & NSS
–	MET (0.060)	–	NL (0.063)
–	P (0.040)		
–	EG (0.013)		
o_6 – correct_course	(1.113)	ds_6 – altitude_command	(1.063)
• ISC & AO		•	ADT & NSS
–	MET (0.060)	–	NL (0.063)
–	P (0.040)		
–	EG (0.013)		
o_7 – correct_altitude	(1.113)	ds_7 – actual_course	(1.016)
• ISC & AO		•	PT & NSS
–	MET (0.060)	–	NL (0.016)
–	P (0.040)		
–	EG (0.013)		
		ds_8 – actual_altitude	(1.016)
		•	PT & NSS
		–	NL (0.016)
		ds_9 – desired_course	(1.032)
		•	PT & SS
		–	NL (0.032)
		ds_{10} – desired_altitude	(1.032)
		•	PT & SS
		–	NL (0.032)

Table 5.5. Properties and Weights Associated with Autopilot Control System.

In Chapter IV we used Figure 4.1: Autopilot Control System, Figure 4.4: Decomposition of *autopilot_software* and Figure 4.5: Expanded Autopilot Control System to study complexity based strictly on the sum of operators and data streams. We concluded that the complexity of a system, with composite operators, should fall between a minimum and maximum value. We also determined by Axiom 3.11, that the whole is greater than the sum of its parts. The top-level diagram, Figure 4.1, represents the minimum complexity (i.e, DS_{min} – the base part of the system); the expanded diagram, Figure 4.5, represents the maximum complexity (i.e, DS_{max} – the whole system). The substructure, Figure 4.4, represents additional complexity (i.e., DS_{sub} – a system part). If composite operators actually decrease the complexity of our whole system, the sum of the parts (DS_{act}), should be less than the maximum complexity (i.e., $DS_{min} < DS_{act} < DS_{max}$).

$$DS = \sum_{i=1}^{n} o_i \left[dsi\left(o_i\right) * dso\left(o_i\right) \right]$$

$$\begin{aligned} DS_{min} &= o_1[(ds_5 + ds_6) * (ds_9 + ds_{10} + ds_3 + ds_4)] + o2[ds_1 * ds_7] \\ &\quad + o_3[ds_2 * ds_8] + o_4[(ds_3 + ds_4 + ds_1 + ds_8) * (ds_5 + ds_6)] \\ &= 1.061[(1.063 + 1.063) * (1.032 + 1.032 + 1.063 + 1.063)] \\ &\quad + 1.033[1.032 * 1.016] + 1.033[1.032 * 1.016] \\ &\quad + 1.009[(1.063 + 1.063 + 1.016 + 1.016) * (1.063 + 1.063)] \\ &= 9.451 + 1.083 + 1.083 + 8.919 = \boxed{20.536} \end{aligned}$$

$$\begin{aligned} DS_{max} &= o_1[(ds_5 + ds_6) * (ds_1 + ds_2 + ds_3 + ds_4)] + o_2[ds_1 * ds_7] \\ &\quad + o_3[ds_2 * ds_8] + o_5[(ds_3 + ds_4 + ds_7 + ds_8) * (ds_2 + ds_1)] \\ &\quad + o_6[(ds_7 + ds_2) * (ds_5)] + o_7[(ds_8 + ds_9) * ds_6] \\ &= 1.061[(1.063+ 1.063) * (1.032 + 1.032 + 1.063 + 1.063)] \\ &\quad + 1.033[1.032 * 1.016] + 1.033[1.032 * 1.016] \\ &\quad + 1.113[(1.063 +1.063 + 1.016 + 1.016) * (1.032 + 1.032)] \\ &\quad + 1.113[(1.016 + 1.032) * 1.063] + 1.113[(1.016 + 1.032) * 1.063] \\ &= 9.451 + 1.083 + 1.083 + 9.552 + 2.423 + 2.423 = \boxed{26.015} \end{aligned}$$

$$\begin{aligned} DS_{sub} &= o_5[1 * (ds_1 + ds_2)] + o_6[ds_1 * 1] + o_7[ds_2 * 1] \\ &= 1.113[1 * (1.032 + 1.032)] + 1.113[1.032 *1] + 1.113[1.032 *1] \\ &= 2.297 + 1.149 + 1.149 = \boxed{4.595} \end{aligned}$$

$DS_{act} = 20.536 + 4.595 = \boxed{25.131}$

Note: Figure 4.4 contains multiple data streams with EXTERNAL markings (i.e., *actual_course, actual_altitude, rudder_status, elevator_status, course_command,* and *altitude_command).* Those data streams were counted in the top-level structure and are not counted a second time in the substructure. Subsequently, Figure 4.5 contains two hyperedges (i.e. *actual_course,* and *actual_altitude*); they are each counted as one output data stream and counted as two separate input data streams.

This example provides an accurate measurement of complexity using composite operators. Therefore, when composite operators are used in a PSDL model it is necessary to calculate the complexity of each diagram and sum the values to obtain the complexity of the actual system. When no composite operators are used, the complexity of the system is actually represented as the complexity of the top-level diagram (i.e., $DS_{act} = DS_{min}$).

Figure 4.2 – Fish Farm Control System I, and Figure 4.3 – Fish Farm Control System II, are two systems that use no composite operators. Their complexities were also calculated and are provided in Table 5.6 for comparison. The Autopilot Control System is the most complex of the three at 25.131. Part of an intuitive evaluation of prototype systems requires some understanding of the system requirements; those requirements indicate what properties are built into the operators and data streams. Without it, the numbers may not represent your intuitive understanding of the system. We stated in Chapter IV, it is not enough to simply examine the diagram especially when the actual complexities are so close to one another.

FIGURE	**SYSTEM**	**COMPLEXITY (*DS)***
4.1/4.4	Autopilot Control System	25.131
4.2	Fish Farm Control System I	24.973
4.3	Fish Farm Control System II	20.543

Table 5.6. Calculated Complexities of Three Systems.

E. SUMMARY

Horst Zuse and Karin Drabe [Ref. 16] developed a broad software application called Zuse/Drabe Measure Information System (ZD-MIS) that contains an extensive database of software measures. Figures 5.3 and 5.4 are screen shots from ZD-MIS. The application contains multiple tutorials designed to educate the user on software measures and then walk them through a decision process to choose measures that fit the situation, code and data structure. Measures are not restricted to complexity. In the database, you will find ways to measure size, defect density, structure and others. You are free to choose from the database or to tailor measures based on some hybrid. PSDL now has a hybrid measure due, in part, to ZD-MIS.

When building a hybrid measure it is essential units match and that nothing is lost empirically or qualitatively by combining measures. Empirical understanding is undoubtedly the most important factor in defining a measurement of any kind. It was discussed at length throughout these chapters. The complexity measure defined as the Dupont Scale provides a comprehensive account of PSDL computational models represented as an augmented graph in CAPS. This measure was specifically kept as uncomplicated as possible to assist future research in this area and to assist in Major Michael Murrah’s Modified Risk Model.

The simplicity of this measure should not be regarded as inconclusive. It is quite possible that further research will yield additional complexity methods or models. However, the application of other factors, functions or considerations; applied to a base equation (i.e, Equation 5.12), will only change the scale of the measurements, that which is considered to be low, medium or high complexity. When an equation (i.e., Equation 5.12) provides a mapping of an empirical relation to a formal relation (i.e. a homomorphism), it provides an accurate basis of understanding. Moreover, it fulfills what needed to be learned from the measurement and what goals were set out to be accomplished.

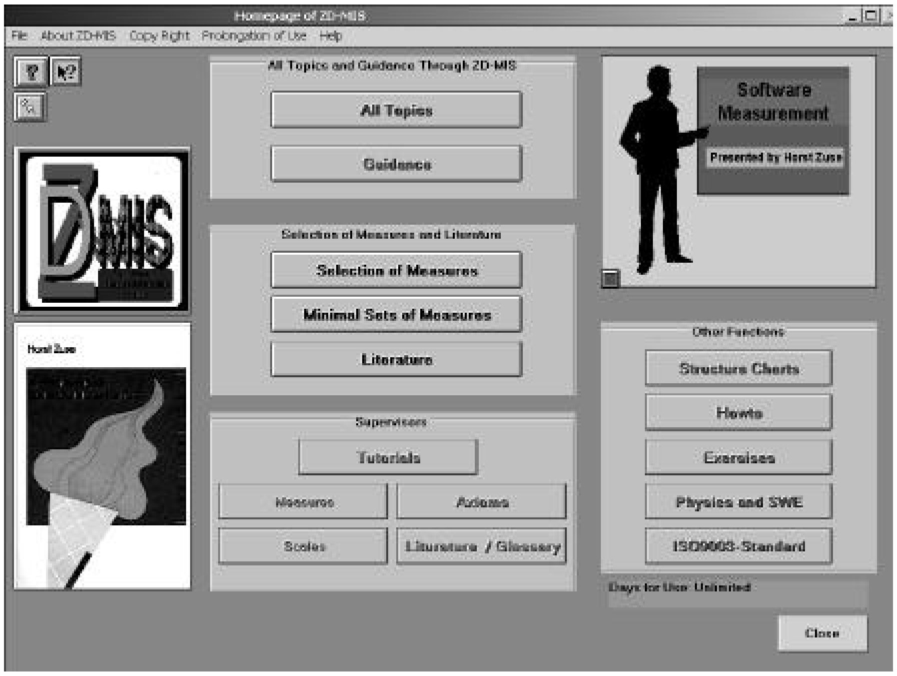


Figure 5.3. ZD-MIS Homepage.

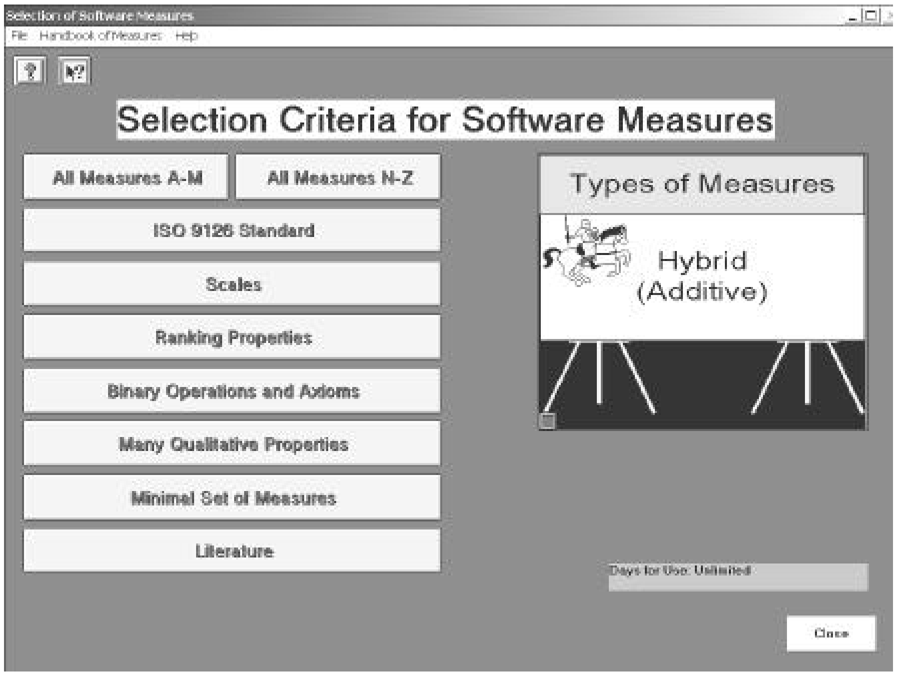


Figure 5.4. Selection Criteria Page of ZD-MIS.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. FUTURE RESEARCH AND CONSIDERATIONS

This thesis was an effort to validate or refine the complexity measure of Nogueira [Ref. 10]. In doing so, the lack of time left me with some open questions that should be explored as future research and/or considerations.

- The weighting tables should be refined to remove conditions that cannot exist in PSDL (eg., it is not possible to have a type operator represented as an External System & Composite – ES & CO).
- My method of developing the weights was only one such method. Others may exist.
- Proof of the equation, possibly by induction.
- Will the use of composite operators lessen the complexity of DS_{max} by a constant factor? If so, the measurement could be calculated entirely from an expanded PSDL source file.
- Using multiple projects and real world examples, validate the measure and define a scale for low, medium and high complexity. It may also be possible to develop a scale using the relational properties of reflexivity, anti-symmetry and transitivity applicable to sets and ordered pairs (i.e., graph theory).
- Build a parser for PSDL files that calculates the complexity.
- Build an analyzer in the CAPS environment that automatically calculates the complexity while developing the prototype.
- Consider what effect hyperedges, which represent global variables, has on the complexity. Hyperedges are not counted multiple times in this measure. For example, there are measures that take into consideration the number of global variables shared by subordinate modules (See Bibliography under Bowles). Thismay provide some answers on how to handle hyperedges represented in PSDL substructures.
- To what degree will the true semantics behind hyperedges (see Figure 6.1) undoubtedly increase the complexity number? The complexity measure under the Dupont Scale represents complexity on a ratio scale. If the increase in complexity due to hyperedges, is a constant, it may simply redefine a low, medium, high scaling. For example, the Autopilot Control System has a complexity of 25.131 on the Dupont Scale; if its hyperedges are considered differently the number may rise to 40.265. As more projects are evaluated, and a scale defined, the Autopilot Control System may come out to be low complexity. That empirical understanding will not change no matter if the number is 25.131 or 40.265.

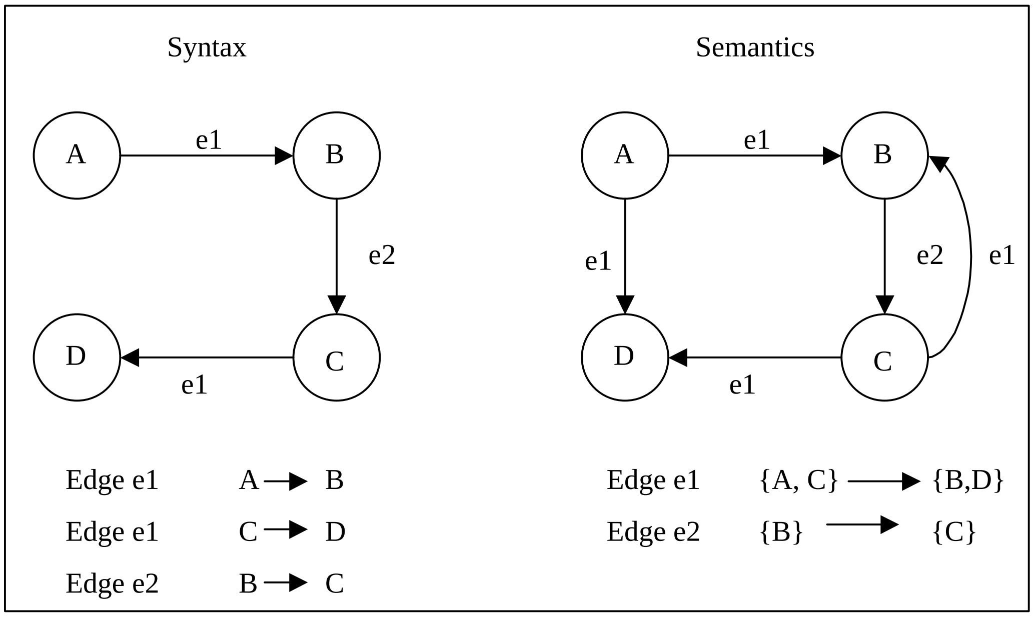


Figure 6.1. PSDL Syntax vs. Semantics.

Figure 6.1 represents the syntax and semantics of hyperedges in PSDL. *A*, *B*, *C*, *D* all represent nodes and *e1* and *e2* represent edges, *e1* being a hyperedge. Both diagrams are representative of the same system. The effect of hyperedges is implicit under Syntax and explicit under Semantics. The diagram under Syntax is what you might expect to see as the dataflow diagram from CAPS, and below it, the PSDL code. Each edge gets listed the number of times it connects operators. The Syntax represents the empirical understanding.

The semantics are quite different. For *e1* there are two producers, *A* and *C* and two consumers, *B* and *D*, labeled as sets. Because *A* produces the data for *e1* it has an effect on both consumers, *B* and *D*; likewise for producer *C*. The Semantics diagram shows this relationship. It contains two additional *e1* edges that are only implicitly defined in the actual diagram. The difference in complexity of this particular example ends up being a factor of two. That means semantically, the system is twice as complex as you would empirically expect. Without further investigation, I am not convinced this an accurate representation of the complexity.

Additionally, to handle hyperedges based on their semantics violates Axiom 3.10:

> Most people cannot manipulate more than a small amount of information at the same time unless there are visual tools available to assist them. Therefore, it is of importance to be able to visualize the process of measurement. It should be easy to present a pictorial representation of the data object and to illustrate graphically the process of application of the metric to a particular data structure.

The explicit representation of the semantics is not intuitive based solely on the dataflow diagram or the PSDL code. It requires further investigation that cannot be easily resolved without redrawing the diagram or a more detailed parser/analyzer.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. COMPLEXITY METRICS FOR DCAPS

SW4510 DCAPS
Complexity Metrics for DCAPS

Final Project Report
26 September 2001

CPT(P) Mike Murrah
MAJ Joe Dupont
LTC Joe Puett

TABLE OF CONTENTS

I. <u>INTRODUCTION</u>

A. Project Aim. Software engineers have long sought to identify, quantify, understand, and control specific aspects of software that directly impact the successful development of software projects. "Size" and "complexity" of software are generally thought to be two of these key aspects; although, there is still considerable debate in the software engineering community as to what is meant by and how to measure “Software Size” and “Software Complexity.” We only have to look back at Fredrick Brook’s seminal work “The Mythical Man-Month” [BROO75] to provide an early analysis of the difficulty in successfully completing a large, complex software project. He outlines the paradox in trying to produce reliable software that is both large and complex because of the many interactions required by an ever increasing number of software modules and by an ever increasing number of people required to produce those modules.

While software size and complexity have been extensively researched, there are still no conclusive complexity metrics that can be calculated very early in the software development cycle that produce reliable measures of the eventual complexity of the delivered software. Even Function Point analysis (while calculable early) has several weaknesses (see Section I.D below). Nogueira [NOGU00] performed one initial investigation of an early calculable complexity measure as an input to his project risk model. However, questions remain about the validity of this measure (see sections IV, VI, & VII). Being able to produce a reliable measure of the eventual complexity of the software early in the software’s design (perhaps during the prototyping phase) is of considerable interest. Early (rather than later) in the development cycle, a software designer has the greatest flexibility in modifying the software design to achieve desired program objectives of cost, time, and functionality. Thus, investigating how and when to obtain early measures of complexity is of significant importance. This project seeks to continue such an investigation.

The specific aim of this project is to examine some existing complexity measures and determine their applicability when applied to the Distributed Computer Aided Prototyping System (DCAPS) environment. Specifically, we will:

- Develop a Prototype System Description Language (PSDL) Analyzer Tool which will calculate side-by-side complexity measures for Nogueira’s Large Grain Complexity (LGC) [NOGU00] and McCabe’s Cyclomatic Complexity metric (MCC) [WATS97] (see Appendices B through E).
- Compare the LGC & MCC complexities of several available PSDL models (see Section VI).

- Attempt to arrive at some conclusions regarding what complexity measures might be best suited for the DCAPS environment (see Section VII).

B. Computer Aided Prototyping System (CAPS). Luqi and Ketabchi introduced CAPS in 1988 [LUQI88] as a means of improving embedded real-time software development thorough the use of tools supporting a two phased approach of rapid prototyping via specification and reusable components followed by automatic program generation. CAPS itself is supported by a specification language known as the Prototype System Description Language (PSDL).

A CAPS prototype is initially instantiated as an augmented dataflow diagram that is then translated into PSDL so that executable prototypes can be produced. Cordeiro [CORD00] summarizes the support provided by CAPS and the CAPS Prototyping Process as follows:

> "CAPS provides the following kinds of support to the prototype designer:
>
> 1) Timing feasibility checking via the scheduler,
> 2) Consistency checking and some automated assistance for project planning, scheduling, designer task assignment, and project completion date estimation via the Evolution Control System,
> 3) Design completion via the editors, and
> 4) Computer-aided software reuse via the software base.
>
> The basic CAPS Prototyping Process:
>
> 1) Based on requirements, design (or modify) the dataflow diagram for the system.
> 2) Assign all appropriate timing and control constraints to the prototype operators. Assign latencies to data streams (if required).
> 3) Assign data types to all data streams.
> 4) Find (in the software base) or build an implementation module for each user-defined data type and each atomic operator. Modules taken from the software base can be modified after retrieval to suit individual needs.
> 5) Build the prototype's user-interface (if required).
> 6) Translate the CAPS-generated (and user-augmented) PSDL program into (a portion of) the Ada supervisor module.
> 7) Run the CAPS scheduler to generate the static and dynamic schedules. This completes the prototype's Ada supervisor module.

8) Compile the prototype. Note: for successful compilation, particular attention must be paid to the formal parameters of atomic operator implementation procedures created in step 4.
9) Execute, evaluate and modify (if appropriate) the prototype and/or the requirements.
10) Return to Step 1 if prototype modification is required.”

The following is a summary of the PSDL Computational Model as described by Luqi, Berzins, and Yeh in [LUQI88a]:

PSDL is based on a computational model containing OPERATORS that communicate via DATA STREAMS, where each stream carries values of a fixed abstract data type. PSDL contains several pre-defined Abstract Data Types (e.g. float, integer, boolean, etc) as well as providing the user the ability to establish user defined Abstract Data Types. Operators can only gain access to other operators when they are connected via data streams. The PSDL computational model is formally represented as an augmented graph:

$$G = (V,E,T(v),C(v))$$

where:

- V is a set of vertices (v)
- E is a set of edges
- T(v) is the set of timing constraints for each vertex v
- C(v) is the set of control constraints for each vertex v

Each vertex represents an operator and each edge represents a data stream (see Annex F for sample PSDL graphs).

Our investigation to determine appropriate complexity measures for software prototypes relied on calculating the complexity of the Software Prototype as it was generated in PSDL. Both LGC and MCC were calculated by determining the complexity of the underlying augmented graph represented by operators and data streams.

C. Software Size. Calculating complexity by itself is meaningless. For a particular metric to have meaning, it must be related to the production factors associated with software development (e.g. time to develop, resources required, functionality produced). Given the limited time of our investigation, we realized that we would not be able to directly relate our complexity measure against such a production factor. Instead, we chose to relate "complexity" against “Software Size”, a parameter that has been demonstrated to directly bear on the production factors of software development [PRES01]. However, relating “complexity” to “size” introduces its own set of inaccuracies and ambiguities. What exactly is meant by “Software Size?” There are numerous opinions on the issue – for instance, Whitmire [WHIT97] as paraphrased in [PRES01] identifies four different views that can be taken regarding what size means in Object Oriented programming:

> "Size is defined in terms of four views: population, volume, length, and functionality. Population is measured by taking a static count of OO entities such as classes or operations. Volume measures are identical to population measures but are collected dynamically, -- at a given instant of time. Length is a measure of a chain of interconnected design elements (e.g., the depth of an inheritance tree is a measure of length). Functionality metrics provide an indirect indication of the value delivered to the customer by an OO application."

For the purposes of our investigation, we decided to use the number of Lines of Code (LOC) of the expanded PSDL file for our comparison metric related to "Software Size." We were confident that this metric (given a sufficiently large set of PSDL models turned into actual delivered software) would correlate directly with any of the production factors associated software development (e.g. time to develop, resources required, functionality produced).

D. Complexity. Because Software Complexity forms the basis for our investigation, some background on previous research efforts to obtain complexity metrics is in order.

One main and most successful areas of complexity research focused on functional complexity calculated through “Function Points.” Nogueira pointed out in his dissertation [NOGU00] that: “Functional complexity has been studied for years because it correlates highly with effort and risk... Note that functional complexity includes two notions of complexity. First, there is the notion of relational complexity describing the mechanistic view of the system. This notion can be objectively measured. Second, there is a rational notion of complexity that is subjective and depends on cognitive limitations of the observer.”

Functional complexity metrics were first introduced by Albrecht [ALBR79] & [ALBR83] and have been widely used because:

1) they are an early metric and can be calculated during the design phases of the software (as early as the prototyping phase as long as the complete system is being prototyped),

2) they are easy to calculate by simply summing 5 parameters, and
3) they can be easily related to LOC

Calculating Function Points is fairly straightforward. Simply count the number of inputs, outputs, queries, files, and system interfaces required in the system. Classify each as either simple, medium or complex. Depending on the parameter and its complexity, the count is multiplied by a weight factor. Table 1 presents the template for the calculation.

Table 1: Function Points Calculation [ALBR83] as presented in [NOGU00]

	Simple	Weight	Medium	Weight	Complex	Weight	Total
Inputs	(	* 3) +	(	* 4) +	(	* 6) =	
Outputs	(	* 4) +	(	* 5) +	(	* 7) =	
Queries	(	* 3) +	(	* 4) +	(	* 6) =	
Files	(	* 7) +	(	* 10) +	(	* 15) =	
Interfaces	(	* 5) +	(	* 7) +	(	* 10) =	
					NAFP	=	Σ

The result of the total is called Non-Adjusted Function Points (NAFP). Next, the user answers 14 questions on a scale of 0 to 5 (0=unimportant to 5=absolutely essential). These 14 questions are [PRES01]:

1) Does the system require reliable backup and recovery?
2) Are data communications required?
3) Are there distributed processing functions?
4) Is performance critical?
5) Will the system run in an existing, heavily utilized opera5ional environment?
6) Does the system require on-line data entry?
7) Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8) Are the master files updated on-line?
9) Are the inputs, outputs, files or inquiries complex?
10) Is the internal processing complex?
11) Is the code designed to be reusable?
12) Are conversion and installation included in the design?
13) Is the system designed for multiple installations in different organizations?
14) Is the application designed to facilitate change and ease of use by the user?

Finally the Function Points are calculated by the formula:

$FP = NAFP * (0.65 + 0.01 * \Sigma F_i)$
where NAFP is the non adjusted Function points
F_i is the answers to each of the fourteen questions

Nogueira [NOGU00] points out that while this approach is attractive, approach, it has many weaknesses: 1) the metric was derived from a study of MIS projects in the seventies and does not account for recursive functions, reuse, inheritance, communication by messages and polymorphism, 2) languages have evolved and differ a lot from the COBOL of the seventies, and 3) programming styles have suffered a dramatic change that is not reflected in the metric.

Kemerer [KEME93] and Kitchenham [KITC93] & [KITC97] also additional shortcomings that make them unsuitable for our investigation:

1) Individual function point elements lack independence
2) Many function point elements were not related to effort required to produce the software.
3) Prediction metrics based on inputs and outputs provided as good a predictor as Function Points.
4) Prediction metrics based on the number of files and the number of outputs was only slightly worse that Function Points.

Additionally, function points would be difficult to calculate within CAPS because many of the inputs are unknown at that stage of prototyping (e.g. queries, files). So, even though Function Points remain as the most common prediction metric, our investigation requires a different approach.

II. PROJECT SCOPE.

A. Scope. Because of the limited length of time and numbers of people available to us to work on this project we limited our scope:

1) We focused our investigation on only two complexity measures McCabe's and Nogueira's. After briefly considering Function Point analysis (as in section I.D above) we decided that it would not provide a meaningful comparison. We also discarded Halstead's Complexity measure because its inputs and outputs were not applicable to the CAPS environment.
2) We compared McCabe's Cyclomatic Complexity Measure & Nogueira's Large Grain Complexity Measure against LOC of

expanded PSDL files instead of comparing the metrics against final production factors of the delivered code.

3) We used a limited number of expanded PSDL files available from previous CAPS projects.

B. Possible sources of error based on Scope. These limitations on scope might limit the accuracy of our conclusions in the following ways:

1) By only examining McCabe's & Nogueira's complexity metrics in detail, we can make no conclusions about the validity of other complexity metrics.
2) By relating the complexity metrics to LOC of the prototype, we are introducing inaccuracies if they are further extrapolated to the impact on software production factors. A better approach would have been to directly compare the complexity measures against the actual effort, budget, time, and functionality produced in delivered software.
3) Because we used a "convenience sample" of available PSDL files from student projects, we can be fairly certain that these data points are <u>not</u> independent (since they were produced by users of similar abilities, within similar time scales, on projects of similar size). Thus, the applicability of our conclusions to the entire possible range of projects for which CAPS may be used is not as strong as it would be had we had a larger and more diverse sample set.

III. <u>METHODOLOGY.</u>

A. Overview. Our team applied the following methodology to this project:

1) Research various complexity measures (see sections I, IV, and V).
2) Identify complexity measures which can be calculated within a DCAPS environment (see sections I, IV, and V)
3) Replicate Nogueira's PSDL Complexity analyzer [NOGU00] (see appendices B to E).
4) Improve the PSDL Complexity Analyzer to include new complexity measures (see appendices B to E).
5) Identify numerous candidate CAPS projects and calculate their complexity metrics.
6) Perform a comparison of complexity metrics based on the sample(see sections VI and VII).

B. Shortcomings. Because of the limitations required by the scope of this project (see section II above), we discovered the following shortcomings of our methodology:

1) We were only able to investigate a limited number of complexity metrics (Nogueira's, McCabe's, Function Points, Halstead's). Of these, we only fully investigated Nogueira's and McCabe's.
2) Replicating Nogueira's analyzer tool proved more difficult than first anticipated because the code of the tool was not available and the tool (as presented in his dissertation) produced invalid results based on his LGC formula.
3) The number and variety of PSDL files available for comparison were severely limited. Perhaps the greatest criticism of our work stems from the lack of independence of this sample set. We found it impossible to make strong conclusions about the applicability of the complexity measures against the entire population of possible CAPS projects because our sample set was of such a small cross-section of possibilities (both in size and function).
4) Our comparison was only between metrics of prototypes and not against the production factors of actual, delivered software. Given a lot more time and resources a better approach would have been to compare the early complexity measures against actual production factors (development time, effort, cost, functionality) of fully delivered software.

IV. <u>NOGUEIRA'S COMPLEXITY MEASURE</u>.

A. Background. Nogueira's Large Grain Complexity metric [NOGU00] emphasizes the "relational" notion of complexity. A relational complexity of an object is a function of the relationships among the components of the object. Meyers [MYER76] identified three factors in measuring complexity of Object Oriented systems:

- Independence: The independence of each component can reduce the complexity of the system if the components are a partition of the system (high cohesion, low coupling).
- Hierarchy: Hierarchical structures allow the stratification of the system in different layers of abstraction.
- Explicit communication: The components should communicate with explicit protocols avoiding any hidden side effects.

B. Metrics for Complexity. While there are a number of reasons why a particular software component might fail, complexity of the component is obviously a significant contributing factor. As Brooks pointed out [BROO76], complexity also directly impacts the length of time required to produce the software and is directly tied to whether the project can be completed at all. Thus, Nogeria focused a portion of his investigation of "risk" into trying quantify an early measure of software complexity. He chose to attempt to measure this complexity at the prototyping stage.

He proved that the specifications written in PSDL can be analyzed to compute their complexity. As discussed above (section I.C), CAPS and PSDL rely on the following: types, operators, data streams and constraints. Types are declarations of abstract data types required for the system. Operators and data streams are the components of a dataflow graph. Finally, constraints represent the real-time constraints that the system must support. All of these combined can be represented in an augmented graph from which the complexity can be calculated.

C. LGC. Nogueira identified two complexity metrics for PSDL: the Fine Granularity Complexity Metric (FGC), and the Large Granularity Complexity Metric (LGC). He chose to compute two different metrics because they indicate two classes of threats when considering complexity from a risk viewpoint. First, he felt that it was important that a software designer be cognizant of operators that are too complex. High complexity on an operator could be corrected by further decomposition. LGC satisfied his need for a metric that computes the total complexity of the system.

Nogueira's FGC expresses the relational complexity of each operator in the PSDL system model. It is calculated by summing the inputs and outputs of all data streams associated with the operator:

$$\text{FGC} = \text{fan-in} + \text{fan-out}$$

Initially, the PSDL Analyzer tool we developed calculated FGC for each operator (see Annex C). However, as our investigation progressed, we eventually removed this functionality because FGC (as intended) provided little information about overall system complexity and thus, was counter to our research goal. We were able to confirm Nogueira's results related to calculating FGC from PSDL files, although the application of the metric remains subjective (i.e. when is FGC too big, when is it just right?).

Nogueira's LGC expresses the relational complexity of the system as a function of the number of operators (O), data streams (D), and types (T).

$$\text{LGC} = O + D + T$$

Within CAPS, system models contain operators that can be layered in a hierarchy. In order to take account of the relational complexity of the entire system, LGC must be calculated by using a flattened hierarchy that contains only leaf nodes. This is accomplished by calculating LGC from the "expanded" PSDL file which fully expands all operators which can be decomposed (Annex B further explains what an expanded PSDL file is and why it is used in the analyzer).

Unfortunately, Nogueira does not provide any explanation related to the logic behind the derivation of his LGC equation. It makes sense that he somehow account for the number of operators and number of data steams, but why should these values be additive? A ratio between the values would provide a better view to the complexity. Also, since data streams are types, why does he add types to the equation a second time? The overall effect is that his LGC metric will continue to grow as system size increases (as system size becomes larger, LGC becomes larger). Nogueira's LGC metric cannot account for a large, yet relatively simple system. A second shortcoming deals with adding types (T) to the equation. Logically, a system which has a large number of instantiated types (large D) but a small number of declared abstract data types (small T) is less complex than a system with large number of types (large T) which are each only instantiated once (large D). Simply adding T to the equation does <u>not</u> account for this relationship. Again a ratio between instantiated types and declared types would have been a better way in which to calculate complexity.

In summarizing Nogueira's LGC metric, it seems that a better relative measure of complexity would have been to establish a ratio between the data-streams and operators and a second ratio between the number of declared types and instantiated data-streams. Thus, a better equation may have been:

$$LGC = w_1(D/O) + w_2(D/T)$$

where w_i are weights associated with the complexity factors

Unfortunately, we did not have time to investigate such a relationship, determine values for the weights, and conclude whether or not such an equation provides a better metric for complexity. Such an investigation was beyond the scope of our project. However, it is an area that provides promise for future research.

V. <u>McCABE'S COMPLEXITY MEASURE</u>.

A. McCabe and Watson provide a good overview of McCabe's complexity metric in a fairly recent NIST special report [WATS96]. There they use control flow graphs to describe the logic structure of software modules. They define a module as a single function or subroutine (in typical languages) that has a single entry and exit point, and is able to be used as a design component via a call/return mechanism. They use nodes to represent computational statements or expressions, and edges represent the transfer of control between nodes.

B. Cyclomatic complexity, v(G). Cyclomatic complexity is defined for each module to be:

e - n + 2
where
e = the number of edges and
n = the number of nodes in the control flow graph

Cyclomatic complexity is known as v(G), where v refers to the cyclomatic number in graph theory and G indicates that the complexity is a function of the graph G. Cyclomatic complexity is a measure of the number of independent paths that exist in a strongly connected, undirected graph (recall that a strongly connected graph is one in which each node is reachable from every other node). It is precisely the minimum number of paths that can, in (linear) combination, generate all possible paths through the module.

It is easy to see the value of cyclomatic complexity for testing of modules. Since the complexity number equals the number of independent paths through the module, it also represents the number of test cases that should be developed to ensure the proper functioning of the module (at least one test case per independent control flow path).

Normally, the cyclomatic number in graph theory is defined as e - n + 1. But McCabe & Watson point out that program control flow graphs are not strongly connected, but can become strongly connected when a "virtual-edge" is added connecting the exit node to the entry node. Thus, the cyclomatic complexity definition for program control flow graphs is derived from the cyclomatic number formula by simply adding one to represent the contribution of the virtual edge.

For our investigation of McCabe's Cyclomatic Complexity metric on CAPS augmented graphs, we liberally applied the same equation and logic. We ignored the directional nature of the data streams, treating directional hyperedges as individual undirected edges and added a virtual edge from the output node to the input node.

C. Metric Range. McCabe established a range of values for what Cyclomatic complexity should be:

Complexity	**Risk**
1-10	A simple program without much risk
11-20	More complex, moderate risk
21-50	Complex, high risk
> 50	Untestable, very high risk

He stated that: "Overly complex modules are more prone to error, are harder to understand, are harder to test, and are harder to modify. Deliberately limiting complexity at all stages of software development, for example as a departmental standard, helps avoid the pitfalls associated with high complexity software."

For the purposes of our investigation, limitations on the range of values for cyclomatic complexity had little value. Our CAPS graphs were not organized as separate modules (although a line of future research might be to examine the benefits of organizing the prototype into decomposable modules of specific size). Thus we were unconcerned with the value of MCC, only with its relative relationship with LOC.

Because the equation for MCC is subtractive [MCC = e - n + 2], we can expect there to be more independence of MCC to LOC than there was with LGC. However, there should not be total independence because as node are added to a diagram there will continue to be multiple edges added. Thus as $n \rightarrow 8$ we find that $LOC \rightarrow 8$, $(e - n) \rightarrow 8$, and thus $MCC \rightarrow 8$. Ideally, we would want a metric were Complexity could stay constant regardless of LOC.

VI. <u>COMPLEXITY COMPARISON</u>.

A. Appendices B through E detail the development of a PSDL analyzer used to calculate complexity of available PSDL files. Annex F provides some sample analysis of 3 of the 30+ files we used in our research. After calculating the LGC and MCC for each file we performed a data comparison to identify any correlations.

B. At first glance, Chart 1 (below) gives you the impression that there is a linear relationship between lines of code and complexity measures.

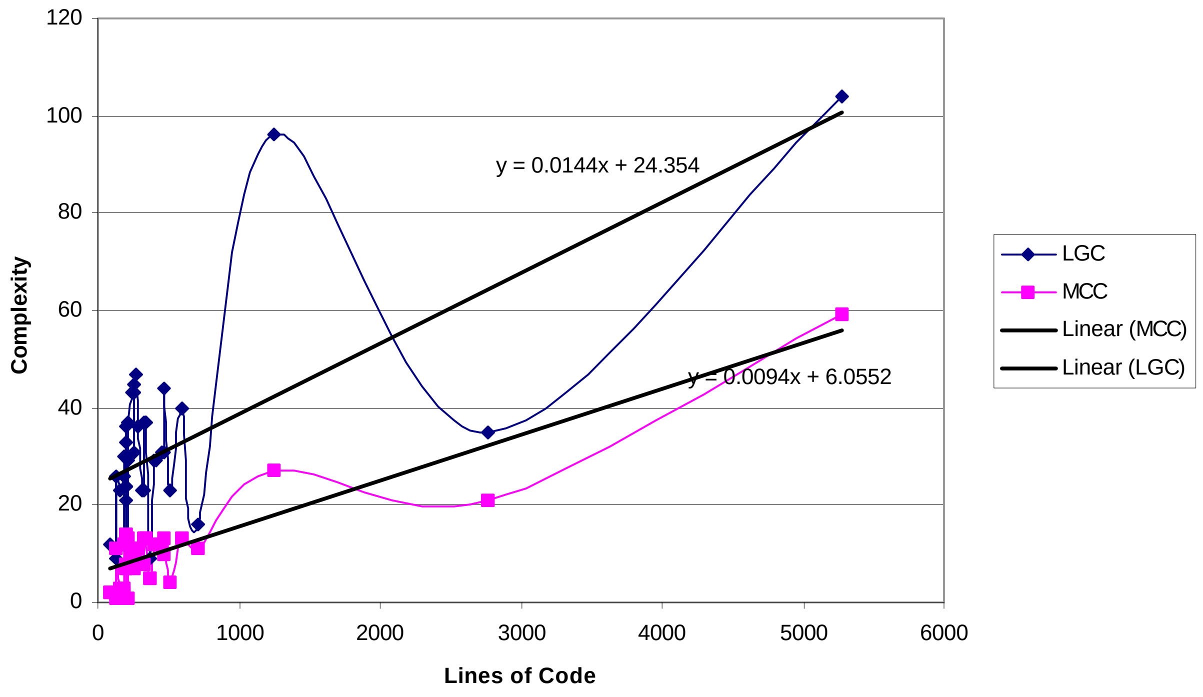


Chart 1: LGC and MCC Plotted against LOC

There is a clear indication that the complexity greatly increases above 1000 lines of code (LOC). Additionally, the fact that both linear trend lines have approximately that same slope, as well as, the similar peaks and valleys shows a correlation between Nogueira's Large Grain Complexity (LGC) measure and McCabe's Cyclomatic Complexity (MCC) measure. Unfortunately, we don't have a clear depiction in this chart because we only have three values in the upper limits. Those values aren't a good representation for larger programs. We do have sufficient data for lines of code less than 750. Changing the scale a bit and dropping the upper values (Chart 2) there is a clearer view of any correlation between the complexity values. The peaks and valleys are equivalent. Note that the slope of the MCC trend line is half that of the LGC trend line. This matches our analysis in section V that predicted that MCC would be more independent of LOC than was LGC, but that both would continue to grow as LOC increased.

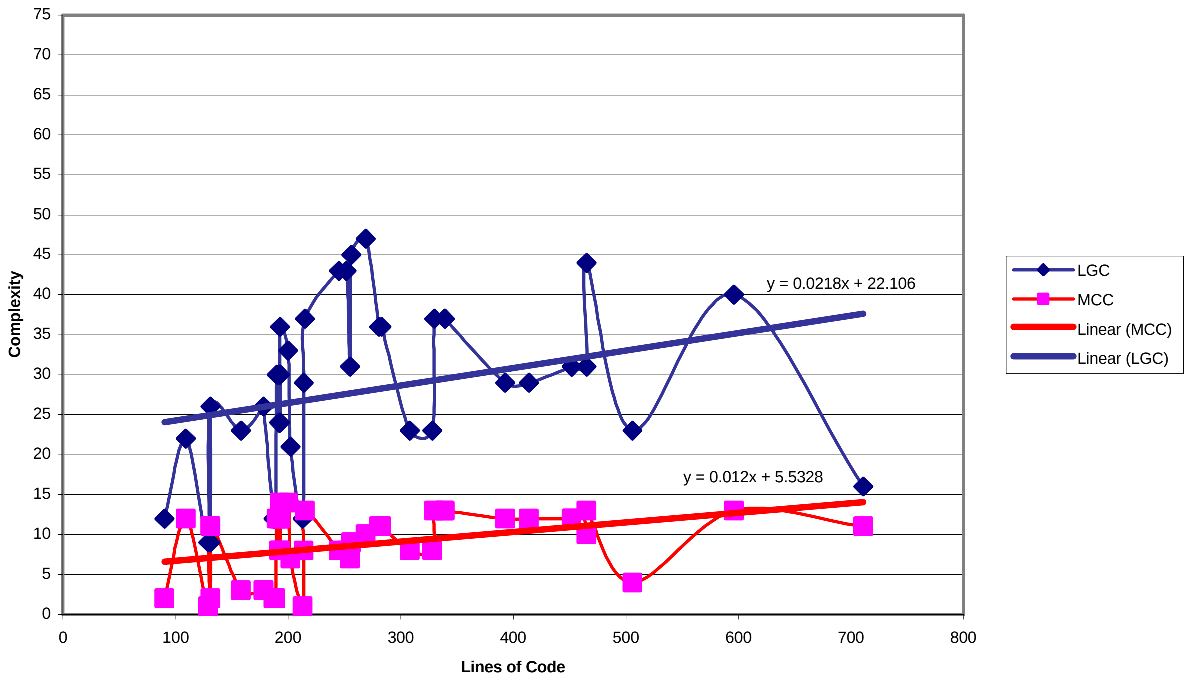


Chart 2: LGC and MCC Plotted against LOC (without last 3 data points)

C. Going yet another step further, (Chart 3) a plot of LGC vs. MCC (Nogueira's Model vs Cyclomatic Complexity Model) shows no injective relationship between the two measures.

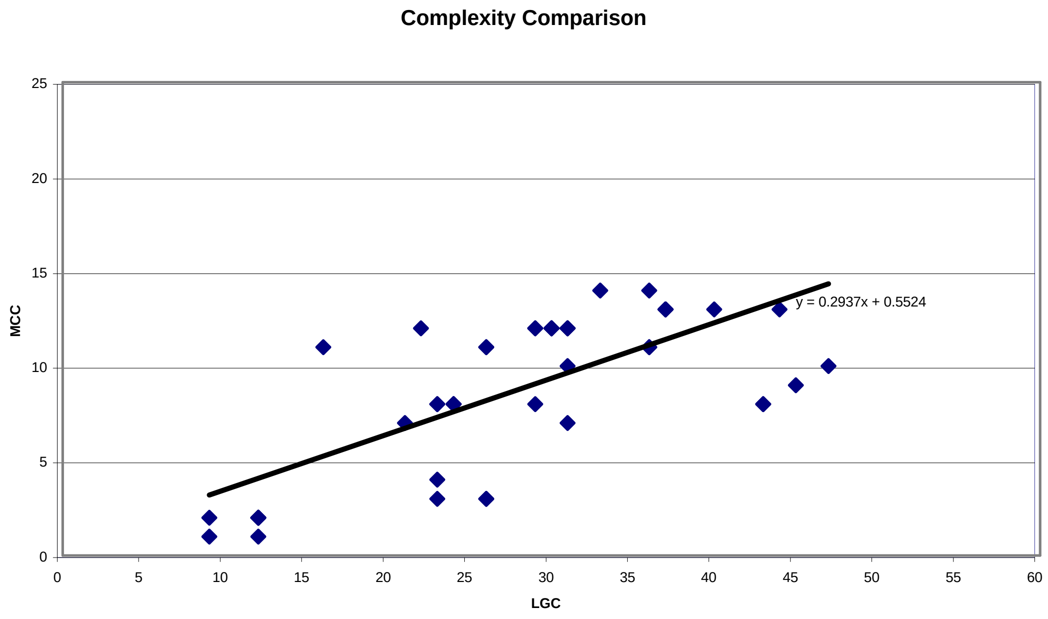


Chart 3: MCC compared to LGC

You will notice that Nogueira's numbers tend to increase over a greater range than the cyclomatic numbers. Adding a linear trend line shows this inaccuracy through its fairly steep slope. A conclusion that is drawn here is that Nogueira's Model is more proportional to total LOC vs. the Cyclomatic Model (again matching our prediction in section V).

VII. <u>CONCLUSIONS.</u>

A. It is very difficult to draw any immediate conclusions that Nogueira's Complexity Model is any better or worse than the widely used and validated Cyclomatic Complexity Model. First, we see some correlation between the two yet Nogueira's model is impacted more by LOC. The sporadic values over the entire range of LOC and the fairly flat slopes indicate there shouldn't be any relationship with LOC. That is to be expected. At first thought you would conclude a program is more complex if it is bigger but you must look at the definitions of complexity. Those definitions use unique data types, operators and edges as their operands; something that is independent of LOC. Secondly, we used a very loose interpretation of MCC to assist us in analyzing PSDL code. We directly applied McCabe's Model to the graphs developed by CAPS. Another interpretation of the Model could have had different results.

B. In summarizing both Nogueira's LGC metric and McCabe's MCC metric, it appears that neither is truly representative of a complexity metric that is independent of program size. Such an independent metric would best suit our needs during prototyping because it would allow us to isolate and correct overly complex portions of the design while ignoring portions that are that are simple. Future work is needed to identify such a metric.

C. Future Research. As a start, it might be worth investigating the metric proposed in section IV:

$$LGC = w_1(D/O) + w_2(D/T)$$

where w_i are weights associated with the complexity factors

Such a metric might prove to be less dependent of LOC than are either Nogueira's LGC and McCabe's MCC.

VIII. LIST OF ANNEXES.

A. References.
B. PSDL Analyzer Algorithm
C. PSDL Analyzer Tool
D. MS Visual Basic Source Code
E. MS Excel Visual Basic for Applications
F. Sample PSDL Graphs and Expanded Source Files

Annex A: References

[ALBR79] Albrecht, A., *Measuring Application Development Productivity.* Proceedings IBM, Oct 1979.

[ALBR83] Albrecht, A. and Gaffney, J., *Software Function Source Lines of Code and Development Effort Prediction,* IEEE Transactions on Software Engineering, SE-9, 1983.

[BROO75] Brooks, F*., The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975.

[CORD00] Cordeiro, M., *Distributed Hard Real-Time Scheduling for a Software Protoyping Environment*, PhD Dissertation, Naval Postgraduate School, Sept 2000

[KEME93] Kemerer, C. *Reliability of Function Points Measurements: A Field Experiment.* Communications of the ACM, Vol 36 No 2. 1993.

[KITC93] Kitchenham, B. and Dansal, K. *Inter-item Correlations among Function Points*. First International Software metric Symposium. IEEE Computer Society press. 1993.

[KITC97] Kitchenham, B. and Linkman, S*. Estimates, Uncertainty, and Risk.* IEEE Software. May-June 1997.

[LUQI88] Luqi and Ketabchi, M., *A Computer-aided Protoyping System*, IEEE Software, 1988.

[LUQI88a] Luqi, Berzins, V., and Yeh, R*., A Prototyping Language for Real-Time Software*, IEEE Transactions on Software Engineering, Vol. 14, No. 10, Oct 1988.

[MYER76] Myers, G., *Software Reliablity*, John Wiley & Sons, 1976.

[NOGU00] Nogueira de León, J. C., *A Formal Model for Risk Assessment in Software Projects*, PhD Dissertation, Naval Postgraduate School, Sept 2000.

[PRES01] Pressman, R. S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill Higher Education, 5th Edition, 2001.

[WHIT97] Whitmire, S., *Object-Oriented Design Measurement*, Wiley, 1997.

[WATS96] Watson, A. and McCabe, T., *Structured Testing: A Testing Methodology using the Cyclomatic Complexity Metric*, NIST Special Publication 500-235, Sept 1996.

Annex B: PSDL Analyzer Algorithm

Overview. The PSDL Analyzer Tool has been developed to quickly and accurately perform analysis on an expanded PSDL file in order to obtain complexity measures from an expanded PSDL file.

Four measures are extracted directly from the PSDL code: uniquely declared data types, edges, vertices (operators), and lines of code. The analyzer accounts for each of these measures by compiling a simple summation of each instance of the measure under consideration.

Logic of the PSDL Analyzer Algorithm. An "expanded" PSDL source file distinguishes itself from a "normal" PSDL source file by removing all of the operator decompositions in the file and deriving an expanded "flattened" file. The fattened file accounts for all operators with their accompanying data streams and data types. The expanded PSDL source file is automatically generated from a translation function during the operation of the CAPS prototyping software. The expanded PSDL source file can be found in its default file location, a "temp" directory in the user's directory structure.

The PSDL analyzer parses the expanded source file seeking out an "OPERATOR" that is identical to the file name of the expanded PSDL source file. Once this operator is accounted for, the analyzer begins to capture the occurrences of the "STATES", "VERTICES", "EDGE", and data type declarations obtained in the "DATA STREAM" portion of the code. Apart from accounting for the total lines of code in the file, the analyzer disregards the remaining code upon reaching "CONTROL CONTRAINTS". This unique combination of identifiers accurately provides the essential information required to obtain the required measurement.

The following table is a subset of an expanded PSDL source file. The names of the expanded file is "fishies.psdl". The total lines of code in the file is 214 (the analyzer does not account for lines in the files which are blank.) The portion of the code below is only the portion that critical information is extracted. In the actual file, the subset of code begins about line 120, for this example we are identifying the first line of code as line 1.

Line 1	`IMPLEMENTATION ADA get_feeding_time_82`
Line 2	`  END`
Line 3	
Line 4	`OPERATOR fishies_53`
Line 5	`  SPECIFICATION`
Line 6	`    STATES feed_schedule: feeding_times INITIALLY`
Line 7	`Empty`
Line 8	`    STATES inlet_valve_position: float INITIALLY 0.0`
Line 9	`    STATES drain_valve_position: float INITIALLY 0.0`
Line 10	`  END`
Line 11	
Line 12	`  IMPLEMENTATION`
Line 13	`    GRAPH`
Line 14	`      VERTEX monitor_nh3_level_55: 80 MS`
Line 15	`      VERTEX monitor_h2o_level_57: 80 MS`
Line 16	`      VERTEX display_status_59: 300 MS`
Line 17	`      VERTEX control_water_flow_61: 100 MS`
Line 18	`      VERTEX monitor_o2_level_63: 80 MS`

```
Line 19         VERTEX adjust_inlet_65: 80 MS
Line 20         VERTEX adjust_drain_67: 80 MS
Line 21         VERTEX control_feeder_69: 100 MS
Line 22         VERTEX get_feeding_time_82
Line 23
Line 24         EDGE o2 monitor_o2_level_63 ->
Line 25   display_status_59
Line 26         EDGE h2o monitor_h2o_level_57 ->
Line 27   display_status_59
Line 28         EDGE nh3 monitor_nh3_level_55 ->
Line 29   display_status_59
Line 30         EDGE o2_status monitor_o2_level_63 ->
Line 31   control_water_flow_61
Line 32         EDGE nh3_status monitor_nh3_level_55 ->
Line 33   control_water_flow_61
Line 34         EDGE h2o_status monitor_h2o_level_57 ->
Line 35   control_water_flow_61
Line 36         EDGE activate_inlet control_water_flow_61 ->
Line 37   adjust_inlet_65
Line 38         EDGE activate_drain control_water_flow_61 ->
Line 39   adjust_drain_67
Line 40         EDGE inlet_setting adjust_inlet_65 ->
Line 41   display_status_59
Line 42         EDGE feeding control_feeder_69 ->
Line 43   display_status_59
Line 44         EDGE feed_schedule get_feeding_time_82 ->
Line 45   control_feeder_69
Line 46         EDGE inlet_valve_position adjust_inlet_65 ->
Line 47   adjust_inlet_65
Line 48         EDGE drain_setting adjust_drain_67 ->
Line 49   display_status_59
Line 50         EDGE drain_valve_position adjust_drain_67 ->
Line 51   adjust_drain_67
Line 52         EDGE feed_schedule get_feeding_time_82 ->
Line 53   get_feeding_time_82
Line 54
Line 55   DATA STREAM
Line 56         o2: float,
Line 57         h2o: float,
Line 58         nh3: float,
Line 59         o2_status: sensor_status,
Line 60         nh3_status: sensor_status,
Line 61         h2o_status: sensor_status,
Line 62         activate_inlet: change_valve,
Line 63         activate_drain: change_valve,
Line 64         inlet_setting: float,
Line 65         feeding: boolean,
Line 66         drain_setting: float
Line 67
Line 68       CONTROL CONSTRAINTS
Line 69         OPERATOR monitor_nh3_level_55
Line 70           PERIOD 1000 MS
```

Aside from counting the total lines of code in the PSDL file, line 4 is the first instance where the analyzer begins to account for the operators, edges, and data types. In line 4, the keyword “OPERATOR” is followed by the PSDL filename “fishies” (the postfix “_53” is ignored). This tells the analyzer that the correct portion of code has been encountered.

In lines 6-9 the analyzer captures the declaration of the data types that are state streams. During this portion of the code, the analyzer identified two unique data type declarations: feeding_times (line 6) and float (lines 8 or 9). Notice that the declaration of "float" is only accounted for one time.

In lines 14-22 the analyzer captures all of the vertices. For the purpose of this documentation we take the liberty of using the words vertices and operators interchangeably. The total vertices for "fishies.psdl" is nine.

Next the analyzer encounters the portion of the PSDL file that contains the edge information. Lines 24-53 encompasses the edge information. The total edge count for "fishies.psdl" is 15. All edges are accounted for. If a hyper-edge has a total of two branches, then the analyzer documents this as two.

Finally, the analyzer enters a section of code that contains the data type declarations. These are separate instances than the previously mentioned state streams and must be accounted for. In this example, lines 56-66 declare additional data types. There is a total of four unique declarations of data types in this section of code. The simple type "float" is instantiated five times but is only declared once. The abstract data type "sensor_status" is instantiated three times but only declared once. "Change_valve" is instantiated twice and counted once. And finally, there is an instantiation of a "boolean" that is only counted once.

The total number of uniquely declared data types then becomes five. Two were identified during the "STATES" portion of the code (lines 6-9) and three additional ones were identified during the previously mentioned portion of code. The table below summarizes the data type declarations.

Line 6	feeding_times	first occurrence
Line 8	float	first occurrence
Line 9	float	repeat (line 8)
Line 56	float	repeat (line 8)
Line 57	float	repeat (line 8)
Line 58	float	repeat (line 8)
Line 59	sensor_status	first occurrence
Line 60	sensor_status	repeat (line 59)
Line 61	sensor_status	repeat (line 59)
Line 62	change_valve	first occurrence
Line 63	change_valve	repeat (line 62)
Line 64	float	repeat (line 8)
Line 65	boolean	first occurrence
Line 66	float	repeat (line 8)

The PSDL analyzer reaches the keywords "CONTROL CONTRAINTS" in line 68 and disregards any additional information it encounters.

To summarize what the results from analyzing this example code, we obtain the following:

Declared Data Types	5
Edges	15
Vertices	9
Lines of code	214

Figure B-1 is the graphical representation of the “fishies.psdl” file. The analyzer does not utilize the PSDL graph, it’s only provided for visual aid in this document. From figure 1 the reader can identify the nine vertices (operators) and the fifteen edges. The graph displays the instantiations of the data types but in isolation does not yield the uniquely declared data types.

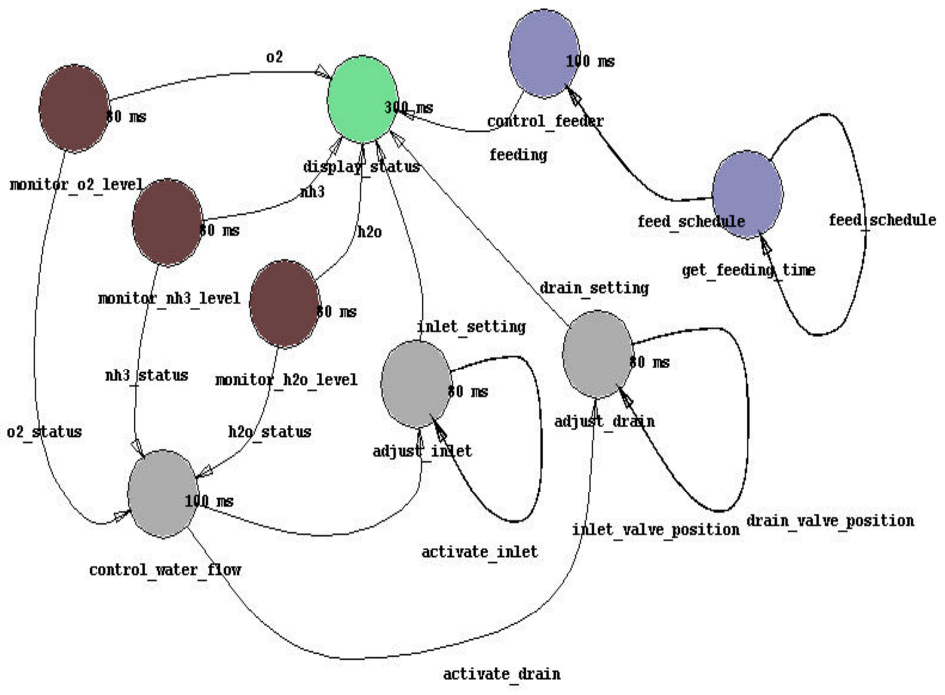


Figure B-1. "fishies.psdl" diagram

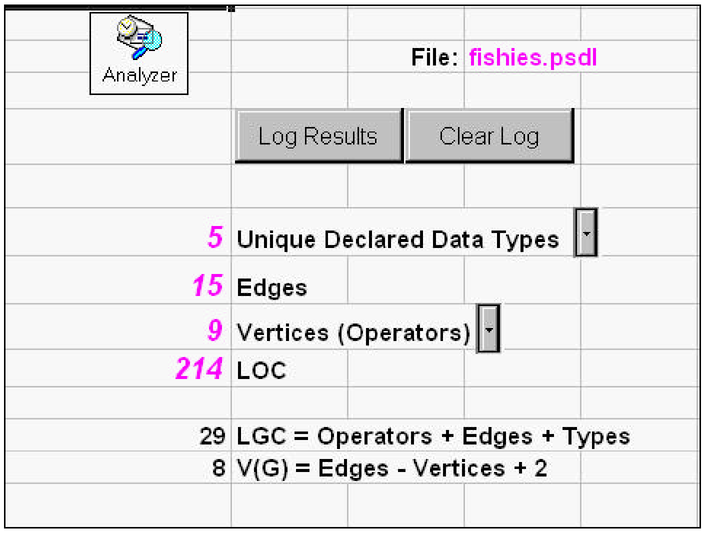
Annex C: PSDL Analyzer Tool

<u>Overview.</u> The PSDL analyzer tool automates the process of extracting the uniquely declared data types, edges, vertices, and lines of code from an expanded PSDL source file. Upon deriving this information, the tool sends the information to the screen, displays the PSDL source code, and affords the user an opportunity to maintain a record. In order to function correctly, a user must have the MS Excel analyzer file and an expanded PSDL source file.

The MS Excel file that is delivered with this documentation is called “PSDL_Analyzer_v3-0.xls”. The excel file contains an embedded executable file that performs the parsing operation on the expanded PSDL file. A user opens the MS Excel file, and then double-clicks the “Analyzer” icon to execute the parsing engine. Be sure to enable all macros and update any links when prompted.

The PSDL analyzer can be operated in two modes: single-file and directory mode. In the single-file mode, individual files are analyzed and the results are made available to the user. In the directory mode, the user can select a directory containing multiple expanded PSDL files, and the analyzer completes analysis on all of the files; generating a log sheet.

<u>Worksheet Orientation.</u> Once the user opens the MS Excel file “PSDL_Analyzer_v3-0.xls” the screen shot in the figure below will be displayed (worksheet “PSDL Analyzer”). The actual data values will vary depending on the last saved version of the MS Excel file. There exist two additional worksheets in this workbook: “PSDL Code” and “Log Sheet”.



File: fishies.psdl

5	Unique Declared Data Types
15	Edges
9	Vertices (Operators)
214	LOC
29	LGC = Operators + Edges + Types
8	V(G) = Edges - Vertices + 2

Users have the ability to view the results of the most recently analyzed expanded PSDL file. This screen shows the results of the PSDL example “fishies.psdl” demonstrated in the previous annex. The four measurements are displayed: unique declared data types, edges, vertices, and lines of code (LOC). Additionally, the name is displayed of the PSDL file. Users have the ability to view each data type and vertices. Select a “drop down” box to view the associated information from the parser.

There exists two command buttons and an embedded macro on the worksheet “PSDL Analyzer”. One command button, “Log Results” takes the results from the most recently executed analysis and places them into the “Log Sheet” worksheet. The second command button, “Clear Log”, removes all entries from the “Log Sheet” worksheet.

The embedded macro “Analyzer” initiates the analyzer engine. This engine contains the algorithms necessary to parse the expanded PSDL files.

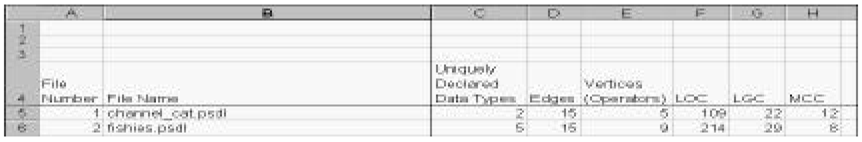
Finally, two calculations are completed and displays: Dr. Nogueira’s Large Granular Complexity and a variation of the McCabe’s Cyclomatic Complexity.

Worksheet “PSDL Code”. When the parser analyzer is invoked, the current PSDL file is read into the MS Excel worksheet and stored in the worksheet “PSDL Code”. Upon each subsequent parser execution, the older PSDL file is removed and replaced by the current file under analysis.

Worksheet “Log Sheet”. The MS Excel worksheet “Log Sheet” maintains a historical log of the files that the analyzer has encountered. In the single-file mode, users must invoke the “Log Results” command button to make an entry into the log. In the directory-mode, entries are made automatically into the log sheet.

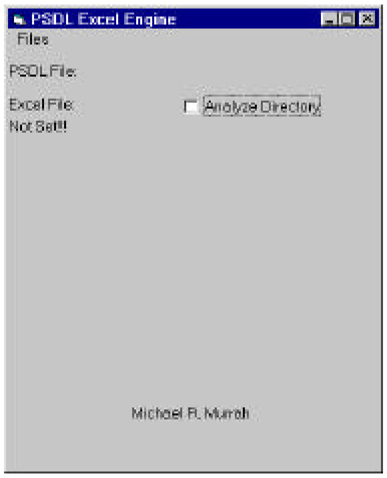
Log Sheet entries will continue to number in an ascending fashion until the contents on the log sheet are cleared. This is completed by invoking the “Clear Log” command button. Caution must be exercised when clearing the log. It is impossible to recover a log once it has been removed. If a user requires removal of a single entry, the user should do so in a manual fashion. Saving the MS Excel file also saves the log.

It is important not to destroy the layout of the log sheet. If it is inadvertently augmented, use the following screen shot to re-establish the template.



	A	B	C	D	E	F	G	H
1								
2								
3								
4	File Number	File Name	Uniquely Declared Data Types	Edges	Vertices (Operators)	LOC	LGC	MCC
5	1	channel_cat.psdl	2	15	5	109	22	12
6	2	fishies.psdl	5	15	9	214	29	8

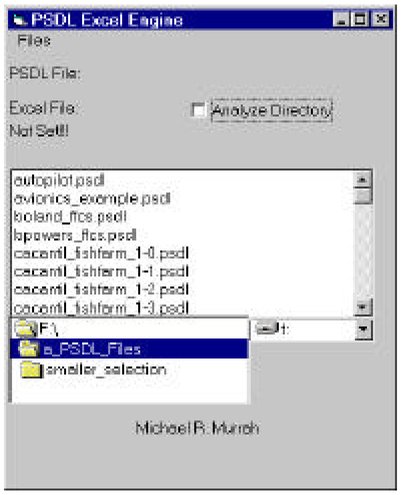
Analyzer Engine. The brains to the PSDL analyzer comes in the form of a parser. The parser was written in MS Visual Basic and interacts with the opened MS Excel file. Users must ensure that the MS Excel file is NOT in the design mode. Double-click on the "Analyze" icon and allow the embedded executable to operate. The following screen should become visible.



Users must interact with the PSDL Analyzer Engine in two ways. First, users must indicate if they intend to analyze single files or whole directories and second, users must establish two file associations.

Checking the box next to "Analyze Directory" tells the analyzer to evaluate an entire directory. Leaving the box unchecked (the default), analyzes single files.

The "Files" menu selection is used to establish the file associations. Two file associations must be established. The first file that is associated is the MS Excel file (the analyzer). This is generally the MS Excel file that you already have open. Simply traverse the directory structure until you located this file and selected it. You will received an appropriate message. Only .xls and .psdl files are visible.



PSDL File:

Excel File:
Not Set!!

☐ Analyze Directory

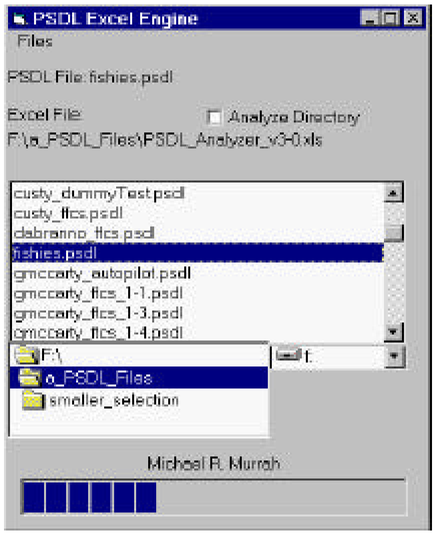
autopilot.psdl
avionics_example.psdl
boland_ffcs.psdl
bpowers_ffcs.psdl
cacantil_fishfarm_1-0.psdl
cacantil_fishfarm_1-1.psdl
cacantil_fishfarm_1-2.psdl
cacantil_fishfarm_1-3.psdl

F:\
a_PSDL_Files
smaller_selection
f:

Michael R. Murrah

The next task is to set the PSDL file. Again traverse the file structure and find the appropriate PSDL file. When the file is selected, the parser carries out either a single-file or directory analysis. A progress meter will give users an indication of the amount of time to complete the action.

Complete the same procedures for a directory analysis. With "Analyze Directory" checked, the parser begins to analyze the directory of the selected file.



PSDL File: fishies.psdl

Excel File:
F:\a_PSDL_Files\PSDL_Analyzer_v3-0.xls

☐ Analyze Directory

custy_dummyTest.psdl
custy_ffcs.psdl
dabranno_ffcs.psdl
fishies.psdl
gmccarty_autopilot.psdl
gmccarty_ffcs_1-1.psdl
gmccarty_ffcs_1-3.psdl
gmccarty_ffcs_1-4.psdl

F:\
a_PSDL_Files
smaller_selection
f:

Michael R. Murrah

Once the parser has completed, refer back to the MS Excel sheet to view the results. If you were analyzing a complete directory, you will have an entry in the "Log Sheet" for each PSDL file in the directory. If only single-file analysis was performed, no entries were made to the "log sheet". However, this can be easily accomplished by invoking the "Log Results" command button. Switch to the "PSDL Code" worksheet to reference the actual expanded PSDL source file.

Known Issues.

The PSDL Analyzer is a working prototype that performs the required functionality but lacks the robustness of production code. These known issues are provided for the user to help minimize or alleviate frustration with its operation.

MS Excel File.

The MS Excel file must be open. The parser engine will not work properly if the associated MS Excel file is not currently open.

When opening the MS Excel file, you must accept all macros and update links.

The MS Excel file's worksheet can not be in the design mode. This will cause a run-time error when the parser is activated.

The MS Excel workbook is not protected. Protecting the workbook creates a run-time error. Users must observe care not to change the contents of the workbook's cells.

Don't move the worksheet template around. The parser is set up to recognize certain cell reference in the MS Excel workbook. If a user desires to change any of the workbook's layout, you must use the "save as" feature in MS Excel and create a duplicate file.

Even though excel file is open, you must still set it with the parser. This is to allow users to create multiple copies of the MS Excel worksheet with different names and store the files in multiple locations.

Parser Engine.

The parser only works correctly on expanded PSDL files. The parser can not distinguish between valid and invalid files. In some cases the parser will execute (seemingly) correct. But the results are not guaranteed unless the source file is expanded.

There is a built in maximum limitation in the parser of 1000 operators and 1000 data types.

Excel file must be opened when running the parser.

Do not add files to the PSDL directory during analysis. The analyzer will not refresh and recognize the added files. If a user needs to add a file in the directory, do so and then manually refresh the directory tree.

Annex D: MS Visual Basic Source Code

The MS Visual Basic Source Code used in the PSDL Analyzer Tool follows:

```
Private Sub Command1_Click(file_size)
' Initialize some variables
char_pos = 1
First_Word = ""
Second_Word = ""
Third_Word = ""
Vertex_Count = 0
Edge_Count = 0
ADT_Count = 0
User_Selected_File = Dir1.Path & "\" & File1.FileName
Const Max_Input = 1000
Lines_of_Code = 0
Excel_Line_Count = 1
right_place = False
still_there = False

    Dim fso As New FileSystemObject, txtfile, _
    fil1 As File, ts As TextStream
    Set fil1 = fso.GetFile(User_Selected_File)


    Dim MyXL As Object   ' Variable to hold reference
                         ' to Microsoft Excel.
    Set MyXL = GetObject(, "Excel.Application")
    Set MyXL = GetObject(Excel_Label.Caption)

    'Clear the PSDL text file
    MyXL.Worksheets("PSDL Code").Columns("A").Clear
    MyXL.Worksheets("PSDL Analyzer").Range("D3") = File1.FileName

    'This array will hold the ADT Names
    Dim ADT() As String
    Dim All_ADT(Max_Input) As String

    'This array will hold the Vertex Names
    Dim Vertex() As String
    Dim All_Vertex(Max_Input) As String

    'Work with the progress bar
    ProgressBar1.Max = file_size
    ProgressBar1.Min = 0

    ' Read the contents of the file.
    Set ts = fil1.OpenAsTextStream(ForReading)
    Do While Not ts.AtEndOfStream
    s = ts.ReadLine
    MyXL.Worksheets("PSDL Code").Range("A" & Excel_Line_Count) = s
    ProgressBar1.Value = Excel_Line_Count

    ' Pick off ANY blank characters
    Do While Mid(s, char_pos, 1) = " "
    char_pos = char_pos + 1
    Loop


 ' ----------------------------------------------------------
    ' Pick off the words
    Do While char_pos <= Len(s)
        ' Get first Word
        If First_Word = "" Then
        Do While (Mid(s, char_pos, 1) <> " ") And _
                (Mid(s, char_pos, 1) <> "")
        First_Word = First_Word + Mid(s, char_pos, 1)
        char_pos = char_pos + 1
        Loop
        Lines_of_Code = Lines_of_Code + 1
        End If

        ' Pick off ANY blank characters
        Do While Mid(s, char_pos, 1) = " "
        char_pos = char_pos + 1
        Loop

        ' Get second Word
        If Second_Word = "" Then
        Do While (Mid(s, char_pos, 1) <> " ") And _
                (Mid(s, char_pos, 1) <> "")
        Second_Word = Second_Word + Mid(s, char_pos, 1)
        char_pos = char_pos + 1
        Loop
        End If

        ' Pick off ANY blank characters
        Do While Mid(s, char_pos, 1) = " "
        char_pos = char_pos + 1
        Loop

        ' Get third Word
        If Third_Word = "" Then
        Do While (Mid(s, char_pos, 1) <> " ") And _
                (Mid(s, char_pos, 1) <> "")
        Third_Word = Third_Word + Mid(s, char_pos, 1)
        char_pos = char_pos + 1
        Loop
        End If

    char_pos = char_pos + 1
    Loop
' ----------------------------------------------------------


    '  Now lets count the Verties, Edges, and Types


    file_name = LCase(Left(File1.FileName, Len(File1.FileName) - 5))
    If First_Word = "OPERATOR" And LCase(Left(Second_Word, Len(file_name))) =
_
        file_name Then
   ' MsgBox "we are in the right place"
    right_place = True
    still_there = True
    End If

    If still_there And First_Word = "CONTROL" And Second_Word = "CONSTRAINTS"
Then
   ' MsgBox "we are leaving"
    right_place = False
    still_there = False
    End If

    If still_there Then
        Select Case First_Word

            ' Edges
            Case "EDGE"
                Edge_Count = Edge_Count + 1

            ' Count the Vertex
            Case "VERTEX"
                All_Vertex(Vertex_Count) = Second_Word
                Vertex_Count = Vertex_Count + 1
            ' Unique Declaration of Data Types in States
            Case "STATES"
                All_ADT(ADT_Count) = Third_Word
                ADT_Count = ADT_Count + 1

            ' Unique Declaration of Data Types in Data Stream

        End Select
        Select Case Right(First_Word, 1)
            Case ":"

                ' Some second words are delimited by ","
                ' Remove the ","
                If Right(Second_Word, 1) = "," Then
                Second_Word = Left(Second_Word, Len(Second_Word) - 1)
                End If

                All_ADT(ADT_Count) = Second_Word
                ADT_Count = ADT_Count + 1
        End Select
    End If



    ' reset the default values for the next text line.
    char_pos = 1
    First_Word = ""
    Second_Word = ""
    Third_Word = ""
    Excel_Line_Count = Excel_Line_Count + 1
    MyXL.Worksheets("PSDL Analyzer").Range("A10") = Lines_of_Code
    Loop
    ts.Close
    ProgressBar1.Visible = False
    ' PSDL_Analyzer.WindowState = vbMinimized


' -----------------------------------------------------------
' -----------------------------------------------------------

' Display the Edge information

'Put edge total in Excel sheet
MyXL.Worksheets("PSDL Analyzer").Range("A8") = Edge_Count

' -----------------------------------------------------------
' Display the ADT information


' ReDim ADT(ADT_Count)
Unique_ADT = All_ADT

For i = 0 To ADT_Count - 1
    temp = Unique_ADT(i)
    For J = i + 1 To ADT_Count - 1
        If Unique_ADT(J) = temp Then
        Unique_ADT(J) = ""
        End If
    Next J
Next i

For i = 0 To ADT_Count - 1
    If Unique_ADT(i) <> "" Then
    Count_ADT = Count_ADT + 1
    End If
Next i

ReDim ADT(Count_ADT)
Counter = 0

'Put ADT total in Excel sheet
MyXL.Worksheets("PSDL Analyzer").Range("A7") = Count_ADT
MyXL.Worksheets("PSDL Analyzer").Abstract_Data_Types.Clear


For i = 0 To ADT_Count - 1
    If Unique_ADT(i) <> "" Then
    ADT(Counter) = Unique_ADT(i)
    Counter = Counter + 1
    End If
Next i

For i = 0 To Count_ADT - 1
    MyXL.Worksheets("PSDL Analyzer").Abstract_Data_Types.AddItem ADT(i)
```

```
Next i

' ------------------------------------------------------------
' Display the Vertex Count

'Put Vertex total in Excel sheet
MyXL.Worksheets("PSDL Analyzer").Range("A9") = Vertex_Count
MyXL.Worksheets("PSDL Analyzer").Vertices.Clear

ReDim Vertex(Vertex_Count)
Vertex = All_Vertex

For i = 0 To Vertex_Count - 1
     MyXL.Worksheets("PSDL Analyzer").Vertices.AddItem Vertex(i)
 Next i
' ------------------------------------------------------------

'Display Line of Code information on Excel sheet
MyXL.Worksheets("PSDL Analyzer").Range("A10") = Lines_of_Code

End Sub

Private Sub Dir1_Change()
    File1.Path = Dir1.Path
    If Excel_Label.Caption = "Not Set!!!" Then
        File1.Pattern = "*.xls"
    End If

End Sub

Private Sub Drive1_Change()
    Dir1.Path = Drive1.Drive
End Sub

Private Sub Exit_Command_Click()
End
End Sub

Private Sub File1_Click()

  If Right(File1.FileName, 3) = "xls" Then
        response = MsgBox("Set " & File1.FileName & _
            " to your Excel file?", vbYesNo, "File Selection")
        If response = vbYes Then   ' User chose Yes.
        Excel_Label.Caption = Dir1.Path & "\" & File1.FileName
        File1.Pattern = "*.psdl"
        MsgBox "Done!"
  End If

    Else

  If Right(File1.FileName, 4) = "psdl" _
          And Excel_Label.Caption = "Not Set!!!" Then
          MsgBox "Please set Your Excel file FIRST"
    Else
  If Right(File1.FileName, 4) = "psdl" _
        And Excel_Label.Caption <> "Not Set!!!" Then

    'response = MsgBox("Analyze " & File1.FileName & " (yes)" _
    &  "  or  directory  "  &  Dir1.Path  &  "  (no)",  vbYesNoCancel,  "File
Selection")

    'If response = vbYes Then   ' User wants a single file
    If Check1.Value = False Then

      file_size = Get_File_Size()
      Name_Label.Caption = "PSDL File: " & File1.FileName
      ProgressBar1.Visible = True
      ProgressBar1.Refresh
      Command1_Click (file_size)
    ' End If ' for the single file selection

    'If response = vbNo Then  'User wants the directory

Else
        ' File1.Pattern = "*.psdl"

        Dim MyXL As Object   ' Variable to hold reference
                    ' to Microsoft Excel.
        Set MyXL = GetObject(, "Excel.Application")
        Set MyXL = GetObject(Excel_Label.Caption)
        Check1.Value = False

        For i = 0 To File1.ListCount - 1
        File1.Selected(i) = False

        Next i

        For i = 0 To File1.ListCount - 1
        File1.Selected(i) = True
        'Check1.Value = True

            MyXL.Worksheets("PSDL Analyzer").checkbox2.Value = True
        Next i

    End If  ' for the directory response

   End If   ' for having a valid PSDL file

End If
End If


End Sub

Private Sub Get_File_Click()

File1.Visible = True
Dir1.Visible = True
```

```
Drive1.Visible = True

End Sub

Function Get_File_Size()
Max_lines_of_Code = 0

    Dim fso As New FileSystemObject, txtfile, _
    fil2 As File, ts As TextStream
    Set fil2 = fso.GetFile(Dir1.Path & "\" & File1.FileName)
    Set ts = fil2.OpenAsTextStream(ForReading)
    Do While Not ts.AtEndOfStream
    s = ts.ReadLine
    Max_lines_of_Code = Max_lines_of_Code + 1
    Loop
    Get_File_Size = Max_lines_of_Code
    ts.Close
End Function

Private Sub Form_Load()
Excel_Flag = False
End Sub

Private Sub Set_File_Click()
Get_File_Click
End Sub
```

Annex E: MS Excel Visual Basic for Applications

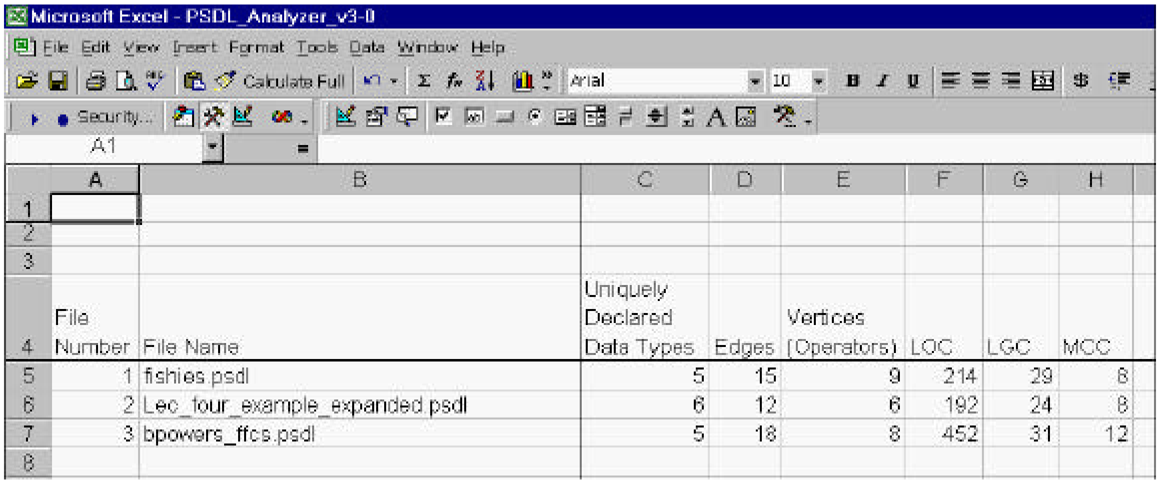
The MS Excel Visual Basic for Applications Source Code used in the PSDL Analyzer Tool follows:

```
Private Sub Abstract_Data_Types_Change()

End Sub

Private Sub CheckBox2_Click()
If CheckBox2.Value = True Then
CheckBox2.Value = False
log_it
End If

End Sub


Private Sub Clear_Log_Click()
Log_Number = 5
response = MsgBox("WARNING! This will erase the contents of the log sheet.
Continue?", vbYesNo)
        If response = vbYes Then
       Do
            Worksheets("Log Sheet").Rows(Log_Number).Clear
            Log_Number = Log_Number + 1
       Loop While Worksheets("Log Sheet").Range("A" & Log_Number) <> ""
        End If
End Sub

Private Sub Single_Log_Click()
If Single_Log.Activate = True Then
MsgBox ("Logging " & Worksheets("PSDL Analyzer").Range("D3"))
log_it
End If
End Sub

Private Sub log_it()

Log_Number = 4

       Do
            Log_Number = Log_Number + 1
       Loop While Worksheets("Log Sheet").Range("A" & Log_Number) <> ""

If Worksheets("Log Sheet").Range("A" & Log_Number) = "" Then
    Worksheets("Log Sheet").Range("A" & Log_Number) = Log_Number - 4
    Worksheets("Log   Sheet").Range("B"   &   Log_Number)   =   Worksheets("PSDL
Analyzer").Range("D3")
    Worksheets("Log   Sheet").Range("C"   &   Log_Number)   =   Worksheets("PSDL
Analyzer").Range("A7")
    Worksheets("Log   Sheet").Range("D"   &   Log_Number)   =   Worksheets("PSDL
Analyzer").Range("A8")
    Worksheets("Log   Sheet").Range("E"   &   Log_Number)   =   Worksheets("PSDL
Analyzer").Range("A9")
```

```
    Worksheets("Log   Sheet").Range("F"   &   Log_Number)   =   Worksheets("PSDL
Analyzer").Range("A10")
    Worksheets("Log   Sheet").Range("G"   &   Log_Number)   =   Worksheets("PSDL
Analyzer").Range("A12")
    Worksheets("Log   Sheet").Range("H"   &   Log_Number)   =   Worksheets("PSDL
Analyzer").Range("A13")
End If

End Sub
```

Annex F: Sample PSDL Graphs and Expanded Source Files

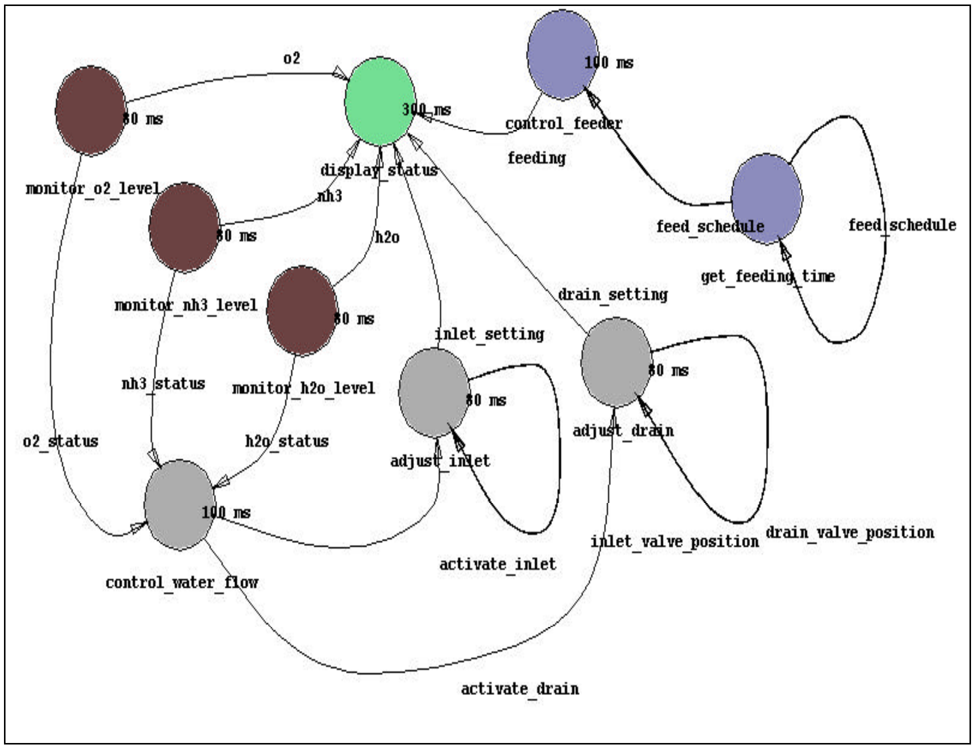
This annex contains analysis of three sample PSDL files (fishies.psdl, Lec_four_example_expanded.psdl, and bpowers_ffcs.psdl) used in our complexity comparisons.



Microsoft Excel - PSDL_Analyzer_v3-0
File Edit View Insert Format Tools Data Window Help
Calculate Full
Arial
10
Security...
A1

File Number	File Name	Uniquely Declared Data Types	Edges	Vertices (Operators)	LOC	LGC	MCC
1	fishies.psdl	5	15	9	214	29	8
2	Lec_four_example_expanded.psdl	6	12	6	192	24	8
3	bpowers_ffcs.psdl	5	18	8	452	31	12

PSDL Graph for Fishies.psdl



Annex F: Sample PSDL Graphs and Expanded Source Files

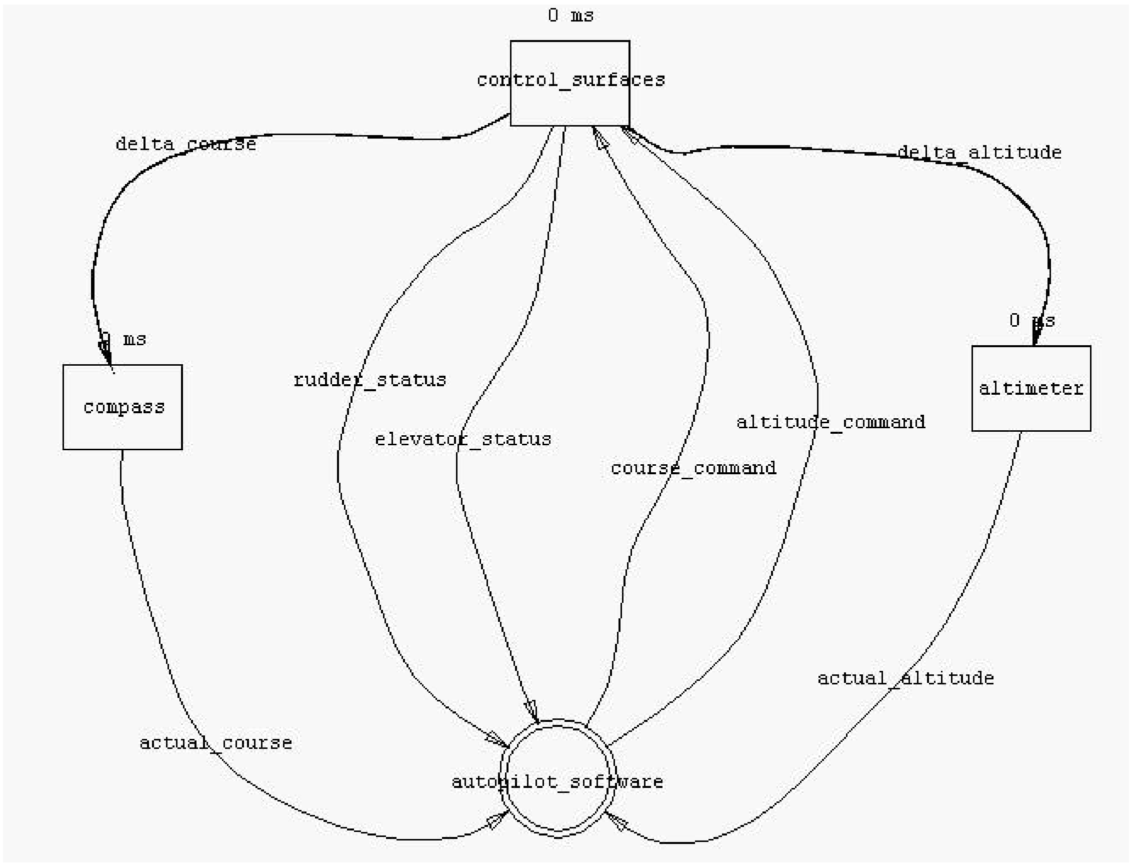
Expanded PSDL File for Fishies.psdl

```
TYPE change_valve
SPECIFICATION
END

IMPLEMENTATION  ADA change_valve
END
TYPE feed_time
SPECIFICATION
END

IMPLEMENTATION  ADA feed_time
END
TYPE feeding_times
SPECIFICATION
  OPERATOR add
  SPECIFICATION
    INPUT
      feed_schedule: feeding_times,
      new_time: feed_time
    OUTPUT
      feed_schedule: feeding_times
  END

  OPERATOR delete
  SPECIFICATION
    INPUT
      feed_schedule: feeding_times,
      new_time: feed_time
    OUTPUT
      feed_schedule: feeding_times
  END

  OPERATOR in_any
  SPECIFICATION
    INPUT
      current_time: military_time,
      feed_schedule: feeding_times
    OUTPUT
      feed_schedule: feeding_times,
      is_in_any: boolean
  END

  OPERATOR empty
  SPECIFICATION
    OUTPUT
      x: feeding_times
  END

END

IMPLEMENTATION  ADA feeding_times
END
```

```
TYPE military_time
SPECIFICATION
END

IMPLEMENTATION  ADA military_time
END
TYPE sensor_status
SPECIFICATION
END

IMPLEMENTATION  ADA sensor_status
END
OPERATOR monitor_nh3_level_55
  SPECIFICATION
    OUTPUT
      nh3: float,
      nh3_status: sensor_status
    MAXIMUM EXECUTION TIME 80 MS
  END

  IMPLEMENTATION ADA monitor_nh3_level_55
  END

OPERATOR monitor_h2o_level_57
  SPECIFICATION
    OUTPUT
      h2o: float,
      h2o_status: sensor_status
    MAXIMUM EXECUTION TIME 80 MS
  END

  IMPLEMENTATION ADA monitor_h2o_level_57
  END

OPERATOR display_status_59
  SPECIFICATION
    INPUT
      o2: float,
      h2o: float,
      nh3: float,
      inlet_setting: float,
      feeding: boolean,
      drain_setting: float
    MAXIMUM EXECUTION TIME 300 MS
  END

  IMPLEMENTATION ADA display_status_59
  END

OPERATOR control_water_flow_61
  SPECIFICATION
    INPUT
      o2_status: sensor_status,
      nh3_status: sensor_status,
      h2o_status: sensor_status
    OUTPUT
      activate_inlet: change_valve,
      activate_drain: change_valve
    MAXIMUM EXECUTION TIME 100 MS
  END

  IMPLEMENTATION ADA control_water_flow_61
  END

OPERATOR monitor_o2_level_63
  SPECIFICATION
    OUTPUT
      o2: float,
      o2_status: sensor_status
    MAXIMUM EXECUTION TIME 80 MS
  END

  IMPLEMENTATION ADA monitor_o2_level_63
  END

OPERATOR adjust_inlet_65
  SPECIFICATION
    INPUT
      activate_inlet: change_valve,
      inlet_valve_position: float
    OUTPUT
      inlet_setting: float,
      inlet_valve_position: float
    MAXIMUM EXECUTION TIME 80 MS
  END

  IMPLEMENTATION ADA adjust_inlet_65
  END

OPERATOR adjust_drain_67
  SPECIFICATION
    INPUT
      activate_drain: change_valve,
      drain_valve_position: float
    OUTPUT
      drain_setting: float,
      drain_valve_position: float
    MAXIMUM EXECUTION TIME 80 MS
  END

  IMPLEMENTATION ADA adjust_drain_67
  END

OPERATOR control_feeder_69
  SPECIFICATION
    INPUT
      feed_schedule: feeding_times
    OUTPUT
      feeding: boolean
    MAXIMUM EXECUTION TIME 100 MS
  END

  IMPLEMENTATION ADA control_feeder_69
  END

OPERATOR get_feeding_time_82
  SPECIFICATION
    INPUT
      feed_schedule: feeding_times
    OUTPUT
      feed_schedule: feeding_times
  END

  IMPLEMENTATION ADA get_feeding_time_82
  END

OPERATOR fishies_53
  SPECIFICATION
    STATES feed_schedule: feeding_times INITIALLY empty
    STATES inlet_valve_position: float INITIALLY 0.0
    STATES drain_valve_position: float INITIALLY 0.0
  END

  IMPLEMENTATION
    GRAPH
      VERTEX monitor_nh3_level_55: 80 MS
      VERTEX monitor_h2o_level_57: 80 MS
      VERTEX display_status_59: 300 MS
      VERTEX control_water_flow_61: 100 MS
      VERTEX monitor_o2_level_63: 80 MS
      VERTEX adjust_inlet_65: 80 MS
      VERTEX adjust_drain_67: 80 MS
      VERTEX control_feeder_69: 100 MS
      VERTEX get_feeding_time_82

      EDGE o2 monitor_o2_level_63 -> display_status_59
      EDGE h2o monitor_h2o_level_57 -> display_status_59
      EDGE nh3 monitor_nh3_level_55 -> display_status_59
      EDGE o2_status monitor_o2_level_63 -> control_water_flow_61
      EDGE nh3_status monitor_nh3_level_55 -> control_water_flow_61
      EDGE h2o_status monitor_h2o_level_57 -> control_water_flow_61
      EDGE activate_inlet control_water_flow_61 -> adjust_inlet_65
      EDGE activate_drain control_water_flow_61 -> adjust_drain_67
      EDGE inlet_setting adjust_inlet_65 -> display_status_59
      EDGE feeding control_feeder_69 -> display_status_59
      EDGE feed_schedule get_feeding_time_82 -> control_feeder_69
      EDGE inlet_valve_position adjust_inlet_65 -> adjust_inlet_65
      EDGE drain_setting adjust_drain_67 -> display_status_59
      EDGE drain_valve_position adjust_drain_67 -> adjust_drain_67
      EDGE feed_schedule get_feeding_time_82 -> get_feeding_time_82

    DATA STREAM
      o2: float,
      h2o: float,
      nh3: float,
      o2_status: sensor_status,
      nh3_status: sensor_status,
      h2o_status: sensor_status,
      activate_inlet: change_valve,
      activate_drain: change_valve,
      inlet_setting: float,
      feeding: boolean,
      drain_setting: float
    CONTROL CONSTRAINTS
      OPERATOR monitor_nh3_level_55
        PERIOD 1000 MS
      OPERATOR monitor_h2o_level_57
        PERIOD 1000 MS
      OPERATOR display_status_59
        PERIOD 1500 MS
      OPERATOR control_water_flow_61
        PERIOD 1000 MS
      OPERATOR monitor_o2_level_63
        PERIOD 1000 MS
      OPERATOR adjust_inlet_65
        TRIGGERED BY SOME activate_inlet
      OPERATOR adjust_drain_67
        TRIGGERED BY SOME activate_drain
      OPERATOR control_feeder_69
        PERIOD 1000 MS
      OPERATOR get_feeding_time_82
  END
```

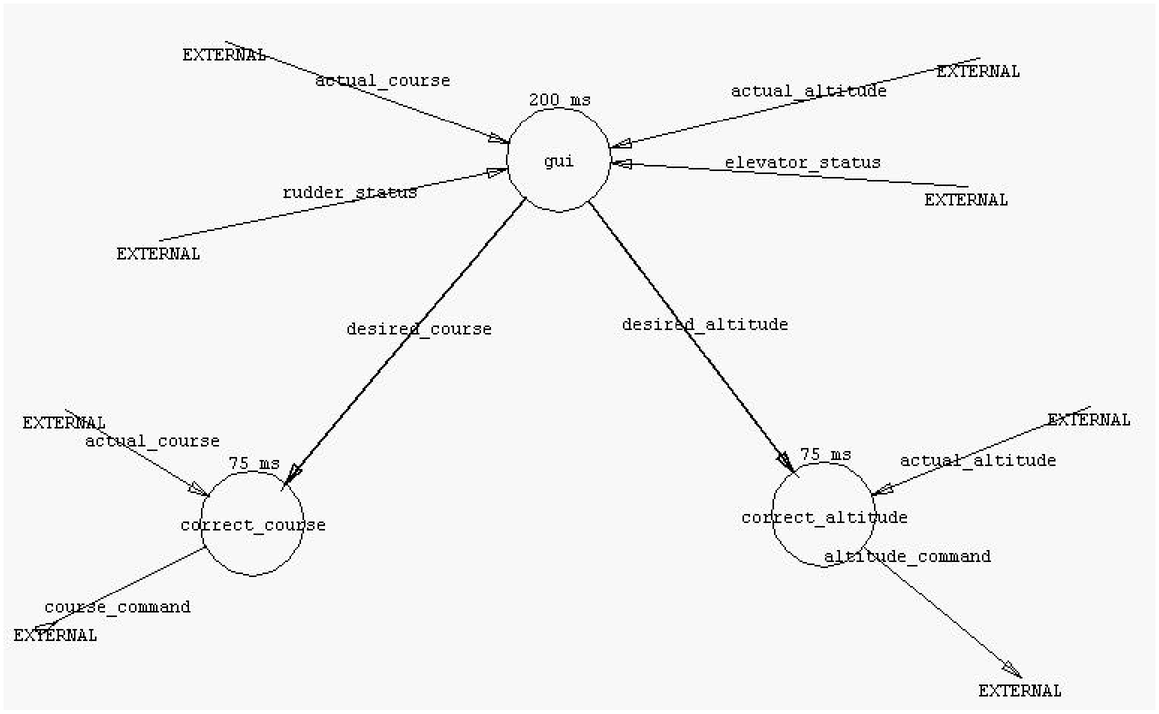
Annex F: Sample PSDL Graphs and Expanded Source Files

PSDL Graph for Lec_Four_Example.psdl



The decomposition of Operator “autopilot_software”.

Annex F: Sample PSDL Graphs and Expanded Source Files



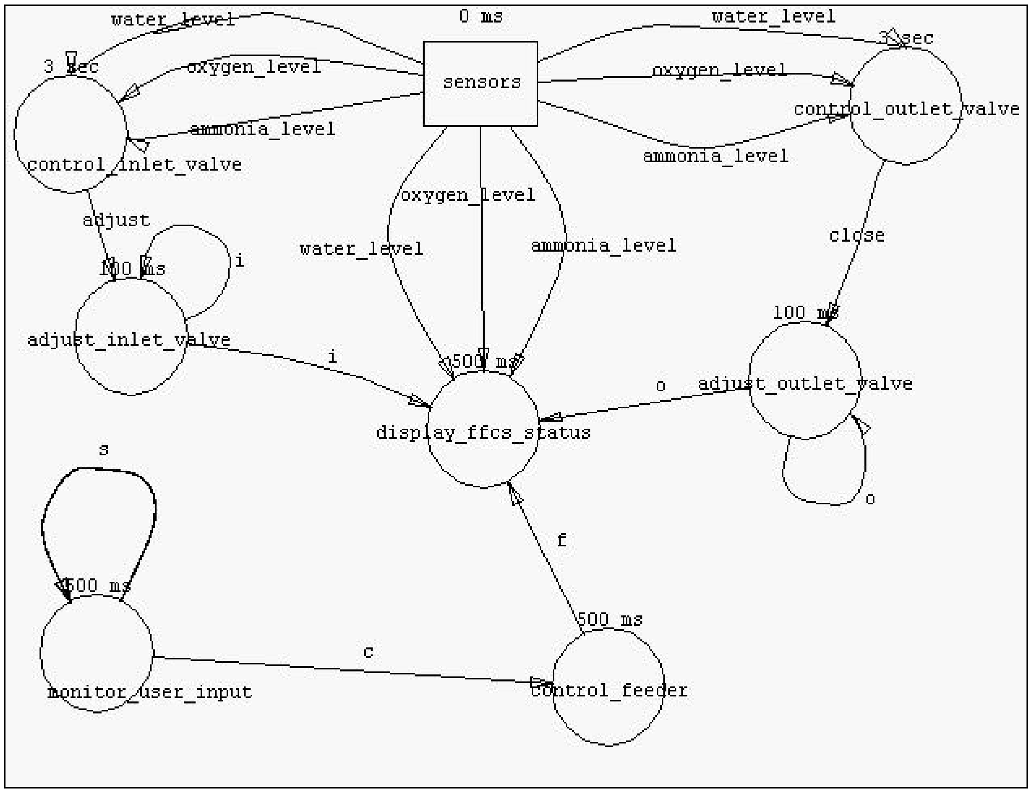
Expanded PSDL File for Lec_Four_Example.psdl

```
TYPE elevator_status_type
SPECIFICATION
END

IMPLEMENTATION ada elevator_status_type
END

TYPE rudder_status_type
SPECIFICATION
END

IMPLEMENTATION ada rudder_status_type
END

TYPE course_command_type
SPECIFICATION
END

IMPLEMENTATION ada course_command_type
END

TYPE altitude_command_type
SPECIFICATION
END

IMPLEMENTATION ada altitude_command_type
END
```

```
OPERATOR compass_7
  SPECIFICATION
    INPUT delta_course: integer
    OUTPUT actual_course: float
    MAXIMUM EXECUTION TIME 0 MS
  END

  IMPLEMENTATION ada compass_7
  END

OPERATOR control_surfaces_10
  SPECIFICATION
    INPUT course_command: course_command_type
    INPUT altitude_command: altitude_command_type
    OUTPUT delta_course: integer
    OUTPUT delta_altitude: integer
    OUTPUT elevator_status: elevator_status_type
    OUTPUT rudder_status: rudder_status_type
    MAXIMUM EXECUTION TIME 0 MS
  END

  IMPLEMENTATION ada control_surfaces_10
  END

OPERATOR altimeter_13
  SPECIFICATION
    INPUT delta_altitude: integer
    OUTPUT actual_altitude: integer
    MAXIMUM EXECUTION TIME 0 MS
  END

  IMPLEMENTATION ada altimeter_13
  END

OPERATOR gui_40
  SPECIFICATION
    INPUT actual_course: float
    INPUT rudder_status: rudder_status_type
    INPUT actual_altitude: integer
    INPUT elevator_status: elevator_status_type
    OUTPUT desired_course: integer
    OUTPUT desired_altitude: integer
    MAXIMUM EXECUTION TIME 200 MS
  END

  IMPLEMENTATION ada gui_40
  END

OPERATOR correct_course_43
  SPECIFICATION
    INPUT desired_course: integer
    INPUT actual_course: float
    OUTPUT course_command: course_command_type
    MAXIMUM EXECUTION TIME 75 MS
  END
  IMPLEMENTATION ada correct_course_43
  END

OPERATOR correct_altitude_46
  SPECIFICATION
    INPUT desired_altitude: integer
    INPUT actual_altitude: integer
    OUTPUT altitude_command: altitude_command_type
    MAXIMUM EXECUTION TIME 75 MS
  END

  IMPLEMENTATION ada correct_altitude_46
  END

OPERATOR lec_four_example_expanded_4
  SPECIFICATION
    STATES delta_course: integer INITIALLY 0.0
    STATES delta_altitude: integer INITIALLY 0
    STATES desired_course: integer INITIALLY 0
    STATES desired_altitude: integer INITIALLY 0
  END

  IMPLEMENTATION
    GRAPH
      VERTEX compass_7_6: 0 MS
        PROPERTY x = 110
        PROPERTY y = 240
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY met_font = 5
        PROPERTY met_unit = 1
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = TRUE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX control_surfaces_10_9: 0 MS
        PROPERTY x = 369
        PROPERTY y = 50
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = -(4)
        PROPERTY met_font = 5
        PROPERTY met_unit = 1
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = TRUE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX altimeter_13_12: 0 MS
        PROPERTY x = 645
        PROPERTY y = 235
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY met_font = 5
        PROPERTY met_unit = 1
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = TRUE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX gui_40_39: 200 MS
      VERTEX correct_course_43_42: 75 MS
      VERTEX correct_altitude_46_45: 75 MS
      EDGE delta_course control_surfaces_10_9 -> compass_7_6
        PROPERTY id = 18
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY latency_font = 5
        PROPERTY latency_unit = 1
        PROPERTY latency_x_offset = 0
        PROPERTY latency_y_offset = -(40)
        PROPERTY spline = "310 84 255 80 199 78 155 86 121 100 104 120
96 141 91 177 "
      EDGE delta_altitude control_surfaces_10_9 -> altimeter_13_12
        PROPERTY id = 20
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY latency_font = 5
        PROPERTY latency_unit = 1
        PROPERTY latency_x_offset = 0
        PROPERTY latency_y_offset = -(40)
        PROPERTY spline = "435 94 463 85 545 90 590 95 623 102 654 130
659 163 651 193 "
      EDGE desired_course gui_40_39 -> correct_course_43_42
      EDGE desired_altitude gui_40_39 -> correct_altitude_46_45
      EDGE actual_course compass_7_6 -> gui_40_39
      EDGE actual_course compass_7_6 -> correct_course_43_42
      EDGE actual_altitude altimeter_13_12 -> gui_40_39
      EDGE actual_altitude altimeter_13_12 -> correct_altitude_46_45
      EDGE elevator_status control_surfaces_10_9 -> gui_40_39
      EDGE rudder_status control_surfaces_10_9 -> gui_40_39
      EDGE course_command correct_course_43_42 -> control_surfaces_10_9
      EDGE      altitude_command      correct_altitude_46_45      ->
control_surfaces_10_9
    DATA STREAM
      actual_course: float,
      actual_altitude: integer,
      elevator_status: elevator_status_type,
      rudder_status: rudder_status_type,
      course_command: course_command_type,
      altitude_command: altitude_command_type
    CONTROL CONSTRAINTS
      OPERATOR compass_7_6
        PERIOD 100 MS
      OPERATOR control_surfaces_10_9
        TRIGGERED BY SOME course_command, altitude_command
        MINIMUM CALLING PERIOD 100 MS
        MAXIMUM RESPONSE TIME 200 MS
      OPERATOR altimeter_13_12
        PERIOD 100 MS
      OPERATOR gui_40_39
        PERIOD 500 MS
        OUTPUT   desired_altitude   IF   ((desired_altitude   >   0)   and
(desired_altitude <= 35000))
        OUTPUT   desired_course   IF   ((desired_course   >=   0.0)   and
(desired_course <= 360.0))
      OPERATOR correct_course_43_42
        TRIGGERED  IF  (((actual_course  -  desired_course)  >  0.5)  or
((actual_course - desired_course) < -(0.5)))
        PERIOD 500 MS
      OPERATOR correct_altitude_46_45
        TRIGGERED  IF  (((actual_altitude  -  desired_altitude)  >  30)  or
((actual_altitude - desired_altitude) < -(30)))
        PERIOD 500 MS
  END
```

Annex F: Sample PSDL Graphs and Expanded Source Files

PSDL Graph for “bpowers_ffcs.psdl”



Expanded PSDL File for bpowers_ffcs.psdl

```
TYPE schedule
SPECIFICATION
END

IMPLEMENTATION ada schedule
END

TYPE command_type
SPECIFICATION
END

IMPLEMENTATION ada command_type
END

TYPE schedule_type
SPECIFICATION

  OPERATOR zero
  SPECIFICATION
    OUTPUT s: schedule_type
  END
```

```
END

IMPLEMENTATION ada schedule_type
END

TYPE continuous_valve_setting_type
SPECIFICATION

  OPERATOR zero
  SPECIFICATION
    OUTPUT o: continuous_valve_setting_type
  END

END

IMPLEMENTATION ada continuous_valve_setting_type
END

TYPE binary_valve_setting_type
SPECIFICATION
END

IMPLEMENTATION ada binary_valve_setting_type
END

OPERATOR control_inlet_valve_7
  SPECIFICATION
    INPUT water_level: float
    INPUT oxygen_level: float
    INPUT ammonia_level: float
    OUTPUT adjust: boolean
    MAXIMUM EXECUTION TIME 3000 MS
  END

  IMPLEMENTATION ada control_inlet_valve_7
  END

OPERATOR adjust_outlet_valve_10
  SPECIFICATION
    INPUT close: boolean
    INPUT o: continuous_valve_setting_type
    OUTPUT o: continuous_valve_setting_type
    MAXIMUM EXECUTION TIME 100 MS
  END

  IMPLEMENTATION ada adjust_outlet_valve_10
  END

OPERATOR control_outlet_valve_13
  SPECIFICATION
    INPUT water_level: float
    INPUT oxygen_level: float
    INPUT ammonia_level: float
    OUTPUT close: boolean
    MAXIMUM EXECUTION TIME 3000 MS
  END

  IMPLEMENTATION ada control_outlet_valve_13
  END

OPERATOR adjust_inlet_valve_16
  SPECIFICATION
    INPUT adjust: boolean
    INPUT i: continuous_valve_setting_type
    OUTPUT i: continuous_valve_setting_type
    MAXIMUM EXECUTION TIME 100 MS
  END

  IMPLEMENTATION ada adjust_inlet_valve_16
  END

OPERATOR display_ffcs_status_19
  SPECIFICATION
    INPUT o: continuous_valve_setting_type
    INPUT i: continuous_valve_setting_type
    INPUT f: binary_valve_setting_type
    INPUT oxygen_level: float
    INPUT ammonia_level: float
    INPUT water_level: float
    MAXIMUM EXECUTION TIME 500 MS
  END

  IMPLEMENTATION ada display_ffcs_status_19
  END

OPERATOR monitor_user_input_22
  SPECIFICATION
    INPUT s: boolean
    OUTPUT c: command_type
    OUTPUT s: boolean
    MAXIMUM EXECUTION TIME 500 MS
  END

  IMPLEMENTATION ada monitor_user_input_22
  END

OPERATOR control_feeder_31
  SPECIFICATION
    INPUT c: command_type
    OUTPUT f: binary_valve_setting_type
    MAXIMUM EXECUTION TIME 500 MS
  END

  IMPLEMENTATION ada control_feeder_31
  END

OPERATOR sensors_37
  SPECIFICATION
    OUTPUT water_level: float
    OUTPUT oxygen_level: float
    OUTPUT ammonia_level: float
    MAXIMUM EXECUTION TIME 0 MS
  END

  IMPLEMENTATION ada sensors_37
  END

OPERATOR bpowers_ffcs_4
  SPECIFICATION
    STATES s: boolean INITIALLY FALSE
  END

  IMPLEMENTATION
    GRAPH
      VERTEX control_inlet_valve_7_6: 3000 MS
        PROPERTY x = 43
        PROPERTY y = 76
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 39
        PROPERTY label_y_offset = 16
        PROPERTY met_font = 5
        PROPERTY met_unit = 2
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = FALSE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX adjust_outlet_valve_10_9: 100 MS
        PROPERTY x = 495
        PROPERTY y = 222
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY met_font = 5
        PROPERTY met_unit = 1
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = FALSE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX control_outlet_valve_13_12: 3000 MS
        PROPERTY x = 557
        PROPERTY y = 59
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY met_font = 5
        PROPERTY met_unit = 2
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = FALSE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX adjust_inlet_valve_16_15: 100 MS
        PROPERTY x = 80
        PROPERTY y = 196
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = -(1)
        PROPERTY label_y_offset = 0
        PROPERTY met_font = 5
        PROPERTY met_unit = 1
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = FALSE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX display_ffcs_status_19_18: 500 MS
        PROPERTY x = 297
        PROPERTY y = 251
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY met_font = 5
        PROPERTY met_unit = 1
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = FALSE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX monitor_user_input_22_21: 500 MS
        PROPERTY x = 59
        PROPERTY y = 384
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 33
        PROPERTY label_y_offset = 21
        PROPERTY met_font = 5
        PROPERTY met_unit = 1
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = FALSE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX control_feeder_31_30: 500 MS
        PROPERTY x = 374
        PROPERTY y = 404
        PROPERTY radius = 35
        PROPERTY color = 62
```

```
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY met_font = 5
        PROPERTY met_unit = 1
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = FALSE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      VERTEX sensors_37_36: 0 MS
        PROPERTY x = 295
        PROPERTY y = 46
        PROPERTY radius = 35
        PROPERTY color = 62
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY met_font = 5
        PROPERTY met_unit = 1
        PROPERTY met_x_offset = 0
        PROPERTY met_y_offset = -(40)
        PROPERTY is_terminator = TRUE
        PROPERTY network_mapping = "local_host"
        PROPERTY criticalness = "hard"
      EDGE water_level sensors_37_36 -> control_inlet_valve_7_6
        PROPERTY id = 39
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 0
        PROPERTY label_y_offset = 0
        PROPERTY latency_font = 5
        PROPERTY latency_unit = 1
        PROPERTY latency_x_offset = 0
        PROPERTY latency_y_offset = -(40)
        PROPERTY spline = "216 19 212 14 169 4 158 4 102 15 90 17 124 6
110 10 70 23 44 40 "
      EDGE oxygen_level sensors_37_36 -> control_inlet_valve_7_6
        PROPERTY id = 41
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 18
        PROPERTY label_y_offset = 12
        PROPERTY latency_font = 5
        PROPERTY latency_unit = 1
        PROPERTY latency_x_offset = 0
        PROPERTY latency_y_offset = -(40)
        PROPERTY spline = "155 26 101 38 "
      EDGE ammonia_level sensors_37_36 -> control_inlet_valve_7_6
        PROPERTY id = 43
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 34
        PROPERTY label_y_offset = 7
        PROPERTY latency_font = 5
        PROPERTY latency_unit = 1
        PROPERTY latency_x_offset = 0
        PROPERTY latency_y_offset = -(40)
```

```
    PROPERTY spline = "168 67 117 74 82 80 "
  EDGE water_level sensors_37_36 -> control_outlet_valve_13_12
    PROPERTY id = 47
    PROPERTY label_font = 5
    PROPERTY label_x_offset = 0
    PROPERTY label_y_offset = 0
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = "392 10 446 14 486 15 556 24 "
  EDGE oxygen_level sensors_37_36 -> control_outlet_valve_13_12
    PROPERTY id = 49
    PROPERTY label_font = 5
    PROPERTY label_x_offset = -(33)
    PROPERTY label_y_offset = 6
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = "429 40 504 40 "
  EDGE ammonia_level sensors_37_36 -> control_outlet_valve_13_12
    PROPERTY id = 51
    PROPERTY label_font = 5
    PROPERTY label_x_offset = -(7)
    PROPERTY label_y_offset = 13
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = "378 70 420 86 454 83 488 71 "
  EDGE adjust control_inlet_valve_7_6 -> adjust_inlet_valve_16_15
    PROPERTY id = 53
    PROPERTY label_font = 5
    PROPERTY label_x_offset = 0
    PROPERTY label_y_offset = 0
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = ""
  EDGE close control_outlet_valve_13_12 -> adjust_outlet_valve_10_9
    PROPERTY id = 55
    PROPERTY label_font = 5
    PROPERTY label_x_offset = -(15)
    PROPERTY label_y_offset = 19
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = "535 114 529 140 "
  EDGE o adjust_outlet_valve_10_9 -> display_ffcs_status_19_18
    PROPERTY id = 57
    PROPERTY label_font = 5
    PROPERTY label_x_offset = 0
    PROPERTY label_y_offset = 0
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = ""
  EDGE i adjust_inlet_valve_16_15 -> display_ffcs_status_19_18
    PROPERTY id = 59
    PROPERTY label_font = 5
    PROPERTY label_x_offset = -(3)
    PROPERTY label_y_offset = 4
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = "172 208 222 220 "
  EDGE f control_feeder_31_30 -> display_ffcs_status_19_18
    PROPERTY id = 61
    PROPERTY label_font = 5
    PROPERTY label_x_offset = 0
    PROPERTY label_y_offset = 0
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = ""
  EDGE c monitor_user_input_22_21 -> control_feeder_31_30
    PROPERTY id = 63
    PROPERTY label_font = 5
    PROPERTY label_x_offset = 0
    PROPERTY label_y_offset = 0
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = ""
  EDGE s monitor_user_input_22_21 -> monitor_user_input_22_21
    PROPERTY id = 65
    PROPERTY label_font = 5
    PROPERTY label_x_offset = 0
    PROPERTY label_y_offset = 0
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
    PROPERTY spline = "99 303 89 277 60 275 44 273 26 298 25 320 "
  EDGE oxygen_level sensors_37_36 -> display_ffcs_status_19_18
    PROPERTY id = 67
    PROPERTY label_font = 5
    PROPERTY label_x_offset = -(18)
    PROPERTY label_y_offset = -(24)
    PROPERTY latency_font = 5
    PROPERTY latency_unit = 1
    PROPERTY latency_x_offset = 0
    PROPERTY latency_y_offset = -(40)
```

```
        PROPERTY spline = ""
      EDGE ammonia_level sensors_37_36 -> display_ffcs_status_19_18
        PROPERTY id = 69
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 13
        PROPERTY label_y_offset = 13
        PROPERTY latency_font = 5
        PROPERTY latency_unit = 1
        PROPERTY latency_x_offset = 0
        PROPERTY latency_y_offset = -(40)
        PROPERTY spline = "348 118 348 156 "
      EDGE water_level sensors_37_36 -> display_ffcs_status_19_18
        PROPERTY id = 71
        PROPERTY label_font = 5
        PROPERTY label_x_offset = -(26)
        PROPERTY label_y_offset = 17
        PROPERTY latency_font = 5
        PROPERTY latency_unit = 1
        PROPERTY latency_x_offset = 0
        PROPERTY latency_y_offset = -(40)
        PROPERTY spline = "238 116 238 154 "
      EDGE i adjust_inlet_valve_16_15 -> adjust_inlet_valve_16_15
        PROPERTY id = 78
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 19
        PROPERTY label_y_offset = 31
        PROPERTY latency_font = 5
        PROPERTY latency_unit = 1
        PROPERTY latency_x_offset = 0
        PROPERTY latency_y_offset = -(40)
        PROPERTY spline = "130 181 142 160 140 141 128 135 118 130 106
130 95 136 91 144 88 153 "
      EDGE o adjust_outlet_valve_10_9 -> adjust_outlet_valve_10_9
        PROPERTY id = 87
        PROPERTY label_font = 5
        PROPERTY label_x_offset = 10
        PROPERTY label_y_offset = 7
        PROPERTY latency_font = 5
        PROPERTY latency_unit = 1
        PROPERTY latency_x_offset = 0
        PROPERTY latency_y_offset = -(40)
        PROPERTY spline = "480 279 489 293 506 296 525 297 534 276 530
248 "
    DATA STREAM
      water_level: float,
      oxygen_level: float,
      ammonia_level: float,
      adjust: boolean,
      close: boolean,
      o: continuous_valve_setting_type,
      i: continuous_valve_setting_type,
      f: binary_valve_setting_type,
      c: command_type
    CONTROL CONSTRAINTS
      OPERATOR control_inlet_valve_7_6
```

```
        PERIOD 10000 MS
        FINISH WITHIN 6000 MS
      OPERATOR adjust_outlet_valve_10_9
        TRIGGERED BY SOME close
        MINIMUM CALLING PERIOD 1000 MS
        MAXIMUM RESPONSE TIME 1000 MS
      OPERATOR control_outlet_valve_13_12
        PERIOD 10000 MS
        FINISH WITHIN 6000 MS
      OPERATOR adjust_inlet_valve_16_15
        TRIGGERED BY SOME adjust
        MINIMUM CALLING PERIOD 1000 MS
        MAXIMUM RESPONSE TIME 1000 MS
      OPERATOR display_ffcs_status_19_18
        PERIOD 1000 MS
        FINISH WITHIN 1000 MS
      OPERATOR monitor_user_input_22_21
        PERIOD 1000 MS
        FINISH WITHIN 1000 MS
      OPERATOR control_feeder_31_30
        PERIOD 1000 MS
        FINISH WITHIN 1000 MS
      OPERATOR sensors_37_36
        PERIOD 10000 MS
  END
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. PSDL AND THE AXIOMS OF CHAPTER III

The following is a comprehensive listing of the axioms presented in Chapter III. This list restates the axioms as described in Chapter III along with an explanation of its relationship to PSDL.

Axiom 3.1: If you prefer a to b, you do not prefer b to a.

Axiom 3.2: If you do not prefer a to b, and do not prefer b to c, then you do not prefer a to c.

Relation to PSDL: There is no direct applicability to PSDL. These were given to position the remaining axioms under the laws of symmetry and transitivity.

Axiom 3.3: Software complexity measures lie on an ordinal or ratio scale.

Relation to PSDL: No matter what the derived complexity measure of PSDL becomes, it must lie on one of these scales. We further stated that a measure on the ratio scale would be best in order for us to understand complexity of PSDL to mean one is twice as complex as another.

Axiom 3.4: Complexity of PSDL, and the augmented graphs associated with it, is implicitly defined as its understandability versus its readability and maintainability.

Relation to PSDL: When deriving complexity measures, there must be an explicit definition of complexity. That definition in this thesis is the level of understandability.

1. Axioms by Tsai, Lopez, Rodriguez and Volovik. [Ref. 14]

Axiom 3.5: "One of the most significant properties of a metric is to conform to intuition. Intuition applied to the objects being measured means that objects, which are seemingly more complex (from one's previous experience) should also be declared as more complex when the metric is applied. Objects that are about equal complexity should also measure about the same. The point is that some objects seem simpler to most people than other objects, and the metric should, in most cases, confirm to this observation."

Relation to PSDL: Initial observation of a PSDL flow graph would indicate equal complexity if there were equal numbers of nodes and edges. That should be the basis of understanding complexity. We find, though, the underlying operator anddata stream properties must also be considered and could possiblycreate flow graphs of unequal complexities.

Axiom 3.6: "Another property of metrics going hand in hand with intuition is consistency (or monotony). In other words, if data structure x is a substructure of a data structure y, then $Complexity(x) \leq Complexity(y)$."

Relation to PSDL: Any substructure (i.e., dataflow diagram) under a composite operator must be less complex than the structure from which it came (i.e., the top-level diagram). Further, if composite operators are used in a PSDL top-level diagram, that diagram will be considered less complex in terms of understandability than similar prototypes that represent the system as a flattened diagram. (See Axiom 3.10 for further clarification).

Axiom 3.7: "The measure should measure the structure of data, not only the size of data. Structure of data tends to be stable during the design process, whereas size of data might not be known even during run-time."

Relation to PSDL: A dataflow diagramrepresents the structure of the PSDL file, and it represents it during the design phase. Additionally, there is not sufficient data to determine if the size of a PSDL file has any correlation on the final size of the actual system. Further, in Chapter I and II, size of PSDL files was determined to be an inconclusive complexity measure. This axiom states it is necessary to look at other things.

Axiom 3.8: "It should be possible to use the metric at a stage of the software design when not all of the decisions have been already made. Measuring a finished product to guide its design is of no use. It is too late. To support these properties, the metric should tolerate incomplete information."

Relation to PSDL: This is the foundation of this research. PSDL is used at the specification and design stages of software development.

Axiom 3.9: "The metric should have the property of automation. Given a data structure description in some formal language, it should be possible to produce a formal machine ready representation of the data structure. The representation seen can be used as an input to a program and the set of measurements can be produced. Resulting measurements themselves should have such a form that humans can easily interpret them, as well as being easily used as an input to some metric-based design support system. To provide automation, the metric should be based on some mathematical foundation."

Relation to PSDL: The driving factor of this research is to validate the complexity measure used by Dr. Nogueira [Ref. 10] as an input to his Risk Management Model. My findings will be incorporated into the dissertation being prepared by Major Michael. In Appendix A, a parser was built that automatically calculated the complexity of PSDL files using MCC and LGC measures. Further work can yield an analyzer provided to CAPS calculating the complexity automatically as the system is built in the drawing pane.

> **Axiom 3.10**: "Most people cannot manipulate more than a small amount of information at the same time unless there are visual tools available to assist them. Therefore, it is of importance to be able to visualize the process of measurement. It should be easy to present a pictorial representation of the data object and to illustrate graphically the process of application of the metric to a particular data structure."

Relation to<u>PSDL:</u> The CAPS process is completely pictorial and represents the PSDL code. This axiom also talks about Chunk Theory [Ref. 9]. The process of building prototypes using CAPS calls for graphs in the drawing pane to consist of 7 ± 2 operators. Further, there is much discussion on the use of compositeoperators to minimize complexity. Thinking of complexity in terms of fan_in and fan_out is easily represented by the graphs.

The use of composite operators can be regarded as the use of nested nodes. Interestingly enough, most authors of complexity measures disagree with this axiom. They describe nested structures as more than complex than sequential structures. Zuse [Ref. 14] describes the following characteristics from authors, L. A. Belady, R. Bache, and P. Piwowarski respectively:

> If we assume that it is more difficult to construct (or understand or maintain) a program whose nodes are imbedded into multiple environments than a program with less nesting, then a weighted summation of the nodes is a reasonable indicator of programming complexity.
>
> …Firstly, if a flowgraph is added by sequencing or nesting, then its metric increases…
>
> Nested control structures are more complex than sequential control structures.

These authors are free to describe their measures in any way they choose so long as they express it. Remember, part of developing a measure is to explain the intuition behind it. We so happen to state that nesting provides less complexity.

2. Axioms by Weyuker [Ref. 14]:

Axiom 3.11: $(\exists P)(\exists Q)\left(P \equiv Q \wedge m(P) \neq m(Q)\right)$

The intuition behind this property is that even though programs compute the same function, it is the details of the implementation that determine the complexity of the program, not the function being computed by the program.

Relation to PSDL: A series of similar CAPS projects, and their generated PSDL code, all having the same requirements and performing the same function, may yield different complexities. The complexity will be dependent on the programmers' interpretation of the requirements and the decisions made on how to build the prototype. Each project will implement similar functions but will exhibit different complexities. The same can be said about any programming language especially in an academic environment. Students of the programming language will be given like projects to complete and not one will be written exactly as another. Moreover, although each program performs similar functionality, the complexity of those programs will be based, partly, on the structure the student decided to use to fulfill the requirements.

Axiom 3.12: $(\forall P)(\forall Q)\ \left(m(P) \leq m(P;Q) \ and\ m(Q) \leq m(P;Q)\right)$

We believe that "montonicity" is another fundamentally important property and it is difficult to imagine the sense in which a measure which fails to satisfy the montonicity property is measuring complexity

Relation to PSDL: Dataflow diagrams with composite operators are normally looked at as two separate diagrams, the top-level structure and the substructure. The composition (i.e., concatenation) of these two diagrams, into one, represents a third <u>expanded diagram</u> of the modeled system. These three are also represented by the PSDL source file and expanded file. This axiom states, the composition (i.e., the expanded diagram) of the two is always greater than the individual parts. Therefore, for every PSDL program, it will hold true that the complexity of any concatenated programs will be more complex or equally complex than any one of its individual programs. Further stated, there is no possible way the concatenation can be less complex.

Axiom 3.13: $(\forall P)(\forall Q)\ \left(m(P) + m(Q) \leq m(P;Q)\right)$

The question is, given that the complexity of a program body should be no less than the complexities or each of its parts, can we make a stronger statement? For example, should the complexity of a program body be no less than the sum of the complexities of its components? Intuitively, in order to implement a program, each of its parts must be implemented.

Relation to PSDL: This axiom is similar to 3.12 but further relates the composition of the two programs to the sum of its parts. For every PSDL program, it will hold true that the complexity of any concatenated program will be more complex or equally complex than the sum of the individual programs. Further stated, the whole is always greater than the sum of its parts. This axiom is overarching basis of determining PSDL complexity.

3. McCabe Cyclomatic Complexity (MCC) Measure

Axiom 3.14:

- MCC-V(G) is the maximum number of linearly independent paths in G; it is the size of a basis set.

Relation to PSDL: This is not necessarily applicable to PSDL. Independent paths do not always represent dataflow in PSDL. Hyperedges may exist in dataflow diagrams showing multiple paths from which data is produced and/or consumed.

- MCC-V(G) depends only on the decision structure of G.

Relation to PSDL: This is not applicable to PSDL. PSDL complexity is based on more than its structure or size.

- MCC-V(G) $\geq$ 1.

Relation to PSDL: PSDL complexity will end up being something greater than one and will never be equal to one.

- G has only one path if and only if v(G) = 1.

Relation to PSDL: Although, PSDL is something greater than one, as a minimum, it is doubtful that level of complexity will ever be reached. That level of complexity may never represent a functional prototype.

- Inserting or deleting functional statements to G does not affect v(G).

Relation to PSDL: This is not applicable to PSDL. Adding additional properties to nodes and edges in PSDL will affect the measure.

- Inserting a new edge in G increases v(G) by unity.

Relation to PSDL: The complexity measure of PSDL is multiplicative and therefore, inserting a new edge will increase the complexity by something greater than one.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. PSDL SOURCE FILE FOR AUTOPILOT CONTROL SYSTEM

```
TYPE rudder_status_type
    SPECIFICATION
    END
    IMPLEMENTATION ada rudder_status_type
    END


TYPE elevator_status_type
    SPECIFICATION
    END
    IMPLEMENTATION ada elevator_status_type
    END


TYPE altitude_command_type
    SPECIFICATION
    END
    IMPLEMENTATION ada altitude_command_type
    END


TYPE course_command_type
    SPECIFICATION
    END
    IMPLEMENTATION ada course_command_type
    END


    OPERATOR autopilot_4
        SPECIFICATION
            STATES delta_course : integer INITIALLY 0
            STATES delta_altitude : integer INITIALLY 0
            DESCRIPTION {}
            AXIOMS {}
        END
      IMPLEMENTATION
      GRAPH
         VERTEX control_surfaces_7_6 : 0 ms
            PROPERTY x = 298
            PROPERTY y = 45
            PROPERTY radius = 35
            PROPERTY color = 62
            PROPERTY label_font = 5
            PROPERTY label_x_offset = 1
            PROPERTY label_y_offset = -15
            PROPERTY met_font = 5
            PROPERTY met_unit = 1
            PROPERTY met_x_offset = 0
            PROPERTY met_y_offset = -40
            PROPERTY is_terminator = true
```

```
   PROPERTY network_mapping = "local_host"
   PROPERTY criticalness = "none"
VERTEX compass_10_9 : 0 ms
   PROPERTY x = 97
   PROPERTY y = 180
   PROPERTY radius = 35
   PROPERTY color = 62
   PROPERTY label_font = 5
   PROPERTY label_x_offset = 0
   PROPERTY label_y_offset = 0
   PROPERTY met_font = 5
   PROPERTY met_unit = 1
   PROPERTY met_x_offset = 0
   PROPERTY met_y_offset = -40
   PROPERTY is_terminator = true
   PROPERTY network_mapping = "local_host"
   PROPERTY criticalness = "hard"
VERTEX altimeter_13_12 : 0 ms
   PROPERTY x = 514
   PROPERTY y = 189
   PROPERTY radius = 35
   PROPERTY color = 62
   PROPERTY label_font = 5
   PROPERTY label_x_offset = 0
   PROPERTY label_y_offset = 1
   PROPERTY met_font = 5
   PROPERTY met_unit = 1
   PROPERTY met_x_offset = 0
   PROPERTY met_y_offset = -40
   PROPERTY is_terminator = true
   PROPERTY network_mapping = "local_host"
   PROPERTY criticalness = "hard"
VERTEX autopilot_software_16_15
   PROPERTY x = 301
   PROPERTY y = 356
   PROPERTY radius = 35
   PROPERTY color = 62
   PROPERTY label_font = 5
   PROPERTY label_x_offset = 1
   PROPERTY label_y_offset = 11
   PROPERTY met_font = 5
   PROPERTY met_unit = 1
   PROPERTY met_x_offset = 0
   PROPERTY met_y_offset = -40
   PROPERTY is_terminator = false
   PROPERTY network_mapping = "local_host"
   PROPERTY criticalness = "none"
EDGE delta_course control_surfaces_7_6 -> compass_10_9
   PROPERTY id = 33
   PROPERTY label_font = 5
   PROPERTY label_x_offset = 0
   PROPERTY label_y_offset = 0
   PROPERTY latency_font = 5
   PROPERTY latency_unit = 1
   PROPERTY latency_x_offset = 0
```

```
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "188 44 149 48 128 59 121 78 "
         EDGE delta_altitude control_surfaces_7_6 -> altimeter_13_12
            PROPERTY id = 35
            PROPERTY label_font = 5
            PROPERTY label_x_offset = -1
            PROPERTY label_y_offset = 0
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "387 46 440 46 486 53 505 92 "
         EDGE actual_course compass_10_9 -> autopilot_software_16_15
            PROPERTY id = 37
            PROPERTY label_font = 5
            PROPERTY label_x_offset = 0
            PROPERTY label_y_offset = 0
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "101 247 103 294 123 330 152 343 210 357
"
         EDGE         actual_altitude         altimeter_13_12         ->
autopilot_software_16_15
            PROPERTY id = 39
            PROPERTY label_font = 5
            PROPERTY label_x_offset = 0
            PROPERTY label_y_offset = 1
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "510 250 510 302 493 334 441 349 395 361
"
         EDGE        rudder_status        control_surfaces_7_6        ->
autopilot_software_16_15
            PROPERTY id = 41
            PROPERTY label_font = 5
            PROPERTY label_x_offset = -17
            PROPERTY label_y_offset = -38
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "218 101 210 129 198 157 196 190 191 227
206 278 "
         EDGE       elevator_status       control_surfaces_7_6       ->
autopilot_software_16_15
            PROPERTY id = 48
            PROPERTY label_font = 5
            PROPERTY label_x_offset = 31
            PROPERTY label_y_offset = 17
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "261 96 243 130 233 183 228 230 267 287 "
         EDGE     altitude_command     autopilot_software_16_15     ->
control_surfaces_7_6
            PROPERTY id = 59
            PROPERTY label_font = 5
            PROPERTY label_x_offset = 30
            PROPERTY label_y_offset = -115
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "335 335 364 301 381 270 393 226 406 177
384 122 367 88 "
         EDGE      course_command      autopilot_software_16_15      ->
control_surfaces_7_6
            PROPERTY id = 66
            PROPERTY label_font = 5
            PROPERTY label_x_offset = -45
            PROPERTY label_y_offset = 25
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "325 318 345 283 363 229 363 181 355 140
341 103 "
      DATA STREAM
         actual_course : real,
         actual_altitude : integer,
         rudder_status : rudder_status_type,
         elevator_status : elevator_status_type,
         altitude_command : altitude_command_type,
         course_command : course_command_type
      CONTROL CONSTRAINTS
         OPERATOR control_surfaces_7_6
            TRIGGERED  BY SOME course_command, altitude_command
            MINIMUM CALLING PERIOD 100 ms
            MAXIMUM RESPONSE TIME 200 ms
            OUTPUT delta_course
                IF TRUE
            OUTPUT delta_altitude
                IF TRUE
            OUTPUT rudder_status
                IF TRUE
            OUTPUT elevator_status
                IF TRUE
         OPERATOR compass_10_9
            PERIOD 100 ms
            OUTPUT actual_course
                IF TRUE
         OPERATOR altimeter_13_12
            PERIOD 100 ms
            OUTPUT actual_altitude
                IF TRUE
```

```
     OPERATOR autopilot_software_16_15
        OUTPUT altitude_command
            IF TRUE
        OUTPUT course_command
            IF TRUE
  END

OPERATOR control_surfaces_7
    SPECIFICATION
        INPUT altitude_command : altitude_command_type
        INPUT course_command : course_command_type
        OUTPUT delta_course : integer
        OUTPUT delta_altitude : integer
        OUTPUT rudder_status : rudder_status_type
        OUTPUT elevator_status : elevator_status_type
        MAXIMUM EXECUTION TIME 0 ms
        DESCRIPTION {}
        AXIOMS {}
    END
  IMPLEMENTATION Ada control_surfaces_7
  END

OPERATOR compass_10
    SPECIFICATION
        INPUT delta_course : integer
        OUTPUT actual_course : real
        MAXIMUM EXECUTION TIME 0 ms
        DESCRIPTION {}
        AXIOMS {}
    END
  IMPLEMENTATION Ada compass_10
  END

OPERATOR altimeter_13
    SPECIFICATION
        INPUT delta_altitude : integer
        OUTPUT actual_altitude : integer
        MAXIMUM EXECUTION TIME 0 ms
        DESCRIPTION {}
        AXIOMS {}
    END
  IMPLEMENTATION Ada altimeter_13
  END

OPERATOR autopilot_software_16
    SPECIFICATION
        INPUT actual_course : real
        INPUT actual_altitude : integer
        INPUT rudder_status : rudder_status_type
        INPUT elevator_status : elevator_status_type
        INPUT altitude_command : altitude_command_type
        OUTPUT altitude_command : altitude_command_type
        OUTPUT course_command : course_command_type
        STATES desired_course : integer INITIALLY 0
        STATES desired_altitude : integer INITIALLY 0
```

```
      DESCRIPTION {}
      AXIOMS {}
  END
IMPLEMENTATION
GRAPH
   VERTEX gui_74_73 : 200 ms
      PROPERTY x = 281
      PROPERTY y = 100
      PROPERTY radius = 35
      PROPERTY color = 62
      PROPERTY label_font = 5
      PROPERTY label_x_offset = 0
      PROPERTY label_y_offset = 0
      PROPERTY met_font = 5
      PROPERTY met_unit = 1
      PROPERTY met_x_offset = 0
      PROPERTY met_y_offset = -40
      PROPERTY is_terminator = false
      PROPERTY network_mapping = "local_host"
      PROPERTY criticalness = "none"
   VERTEX correct_course_77_76 : 75 ms
      PROPERTY x = 194
      PROPERTY y = 310
      PROPERTY radius = 35
      PROPERTY color = 62
      PROPERTY label_font = 5
      PROPERTY label_x_offset = 0
      PROPERTY label_y_offset = 0
      PROPERTY met_font = 5
      PROPERTY met_unit = 1
      PROPERTY met_x_offset = 0
      PROPERTY met_y_offset = -40
      PROPERTY is_terminator = false
      PROPERTY network_mapping = "local_host"
      PROPERTY criticalness = "none"
   VERTEX correct_altitude_80_79 : 75 ms
      PROPERTY x = 398
      PROPERTY y = 305
      PROPERTY radius = 35
      PROPERTY color = 62
      PROPERTY label_font = 5
      PROPERTY label_x_offset = 0
      PROPERTY label_y_offset = 0
      PROPERTY met_font = 5
      PROPERTY met_unit = 1
      PROPERTY met_x_offset = 0
      PROPERTY met_y_offset = -40
      PROPERTY is_terminator = false
      PROPERTY network_mapping = "local_host"
      PROPERTY criticalness = "none"
   EDGE actual_course EXTERNAL -> gui_74_73
      PROPERTY id = 85
      PROPERTY label_font = 5
      PROPERTY label_x_offset = 20
      PROPERTY label_y_offset = 4
      PROPERTY latency_font = 5
      PROPERTY latency_unit = 1
      PROPERTY latency_x_offset = 0
      PROPERTY latency_y_offset = -40
      PROPERTY spline = "101 51 149 55 197 67 "
   EDGE actual_altitude EXTERNAL -> gui_74_73
      PROPERTY id = 90
      PROPERTY label_font = 5
      PROPERTY label_x_offset = -19
      PROPERTY label_y_offset = 0
      PROPERTY latency_font = 5
      PROPERTY latency_unit = 1
      PROPERTY latency_x_offset = 0
      PROPERTY latency_y_offset = -40
      PROPERTY spline = "493 60 434 63 376 73 "
   EDGE rudder_status EXTERNAL -> gui_74_73
      PROPERTY id = 95
      PROPERTY label_font = 5
      PROPERTY label_x_offset = 0
      PROPERTY label_y_offset = 0
      PROPERTY latency_font = 5
      PROPERTY latency_unit = 1
      PROPERTY latency_x_offset = 0
      PROPERTY latency_y_offset = -40
      PROPERTY spline = "107 113 "
   EDGE elevator_status EXTERNAL -> gui_74_73
      PROPERTY id = 100
      PROPERTY label_font = 5
      PROPERTY label_x_offset = 0
      PROPERTY label_y_offset = 0
      PROPERTY latency_font = 5
      PROPERTY latency_unit = 1
      PROPERTY latency_x_offset = 0
      PROPERTY latency_y_offset = -40
      PROPERTY spline = "486 114 "
   EDGE actual_course EXTERNAL -> correct_course_77_76
      PROPERTY id = 105
      PROPERTY label_font = 5
      PROPERTY label_x_offset = 16
      PROPERTY label_y_offset = -3
      PROPERTY latency_font = 5
      PROPERTY latency_unit = 1
      PROPERTY latency_x_offset = 0
      PROPERTY latency_y_offset = -40
      PROPERTY spline = "24 227 "
   EDGE actual_altitude EXTERNAL -> correct_altitude_80_79
      PROPERTY id = 110
      PROPERTY label_font = 5
      PROPERTY label_x_offset = -45
      PROPERTY label_y_offset = -2
      PROPERTY latency_font = 5
      PROPERTY latency_unit = 1
      PROPERTY latency_x_offset = 0
      PROPERTY latency_y_offset = -40
      PROPERTY spline = "566 227 "
```

```
         EDGE desired_course gui_74_73 -> correct_course_77_76
            PROPERTY id = 114
            PROPERTY label_font = 5
            PROPERTY label_x_offset = -35
            PROPERTY label_y_offset = -2
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = ""
         EDGE desired_altitude gui_74_73 -> correct_altitude_80_79
            PROPERTY id = 116
            PROPERTY label_font = 5
            PROPERTY label_x_offset = 30
            PROPERTY label_y_offset = 3
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = ""
         EDGE course_command correct_course_77_76 -> EXTERNAL
            PROPERTY id = 127
            PROPERTY label_font = 5
            PROPERTY label_x_offset = 54
            PROPERTY label_y_offset = -5
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "41 315 37 314 "
         EDGE altitude_command correct_altitude_80_79 -> EXTERNAL
            PROPERTY id = 132
            PROPERTY label_font = 5
            PROPERTY label_x_offset = -31
            PROPERTY label_y_offset = -5
            PROPERTY latency_font = 5
            PROPERTY latency_unit = 1
            PROPERTY latency_x_offset = 0
            PROPERTY latency_y_offset = -40
            PROPERTY spline = "547 321 547 321 553 321 "
      CONTROL CONSTRAINTS
         OPERATOR gui_74_73
            PERIOD 500 ms
            OUTPUT desired_course
                IF desired_course >= 0.0 and desired_course <= 360.0
            OUTPUT desired_altitude
                IF desired_altitude > 0 and desired_altitude <= 35000
         OPERATOR correct_course_77_76
            TRIGGERED
            IF   (((actual_course   -   desired_course)   >   0.5)   or
((actual_course - desired_course) < -(0.5)))
            PERIOD 500 ms
            OUTPUT course_command
                IF TRUE
         OPERATOR correct_altitude_80_79
```

```
            TRIGGERED
            IF   (((actual_altitude   -   desired_altitude)   >   30)   or
((actual_altitude - desired_altitude) < -(10)))
            PERIOD 500 ms
            OUTPUT altitude_command
                IF TRUE
      END

    OPERATOR gui_74
        SPECIFICATION
            INPUT actual_course : real
            INPUT actual_altitude : integer
            INPUT rudder_status : rudder_status_type
            INPUT elevator_status : elevator_status_type
            OUTPUT desired_course : integer
            OUTPUT desired_altitude : integer
            MAXIMUM EXECUTION TIME 200 ms
            DESCRIPTION {}
            AXIOMS {}
        END
      IMPLEMENTATION Ada gui_74
      END

    OPERATOR correct_course_77
        SPECIFICATION
            INPUT actual_course : real
            INPUT desired_course : integer
            OUTPUT course_command : course_command_type
            MAXIMUM EXECUTION TIME 75 ms
            DESCRIPTION {}
            AXIOMS {}
        END
      IMPLEMENTATION Ada correct_course_77
      END

    OPERATOR correct_altitude_80
        SPECIFICATION
            INPUT actual_altitude : integer
            INPUT desired_altitude : integer
            OUTPUT altitude_command : altitude_command_type
            MAXIMUM EXECUTION TIME 75 ms
            DESCRIPTION {}
            AXIOMS {}
        END
      IMPLEMENTATION Ada correct_altitude_80
      END
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Douglas, Bruce Powell, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns,* Addison-Wellesley, New York, NY, 1999.

2. Dupont, MAJ Joe, Murrah, MAJ Mike, Puett, LTC Joe, "Complexity Metrics for DCAPS," final project report for SW 4510, Naval Postgraduate School, Montery, CA, September 2001.

3. Halstead, Maurice H., *Elements of Software Science, Operating, and Programming Systems Series,* Volume 7, Elsevier, New York, NY, 1977.

4. Henry, S. M. and Kafura, D. G., "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering,* Vol. 7, No. 5, pp. 510-518, 1981.

5. "Elementary Concepts in Statistics," [http://www.statsoftinc.com/textbook/esc.html], StatSoft, Inc., 1984-2002.

6. Luqi and Berzins, V., "Rapidly Prototyping Real-Time Systems," *IEEE Software,* pp. 25-36, September 1988.

7. McCabe, T., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 3, pp. 312-327, 1976.

8. "Merriam-Webster Dictionary," [http://www.m-w.com.], Merriam-Webster, Inc, 2002.

9. Miller, G.A., "The Magical Number Seven, Plus or Minus Two: Some Limits On Our Capacity for Processing Information," *Psychological Review,* 1963.

10. Nogueira de León, J. C., *A Formal Model for Risk Assessment in Software Projects,* Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, September 2000.

11. Pressman, R. S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill Higher Education, 5th Edition, 2001.

12. Roberts, Fred S., "Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences," *Encyclopedia of Mathematics and Its Applications*, Addison Wesley Publishing Company, 1979.

13. Whitmire, S., *Object-Oriented Design Measurement*, Wiley, 1997.

14. Zuse, Horst, *Software Complexity – Measures and Methods*, Walter de Gruyter & Co., New York, NY, 1991.

15. Zuse, Horst; *A Framework of Software Measurement*, Walter de Gruyter & Co., New York, NY, 1997.

16. Zuse, Horst, Drabe, Karin, *ZD-MIS, Zuse/Drabe Measurement Information System*, [horst.zuse@t-online.de], Berlin, Germany, 2001.

BIBLIOGRAPHY

Alexander, C., “Notes on the Synthesis of Form,” *Harvard Univ. Press*, Cambridge, MA, 1964.

Basili, Victor, R., Caldiera, Gianluigi, Rombach, Dieter, H., “Software Measurement,” *Encyclopedia of Software Engineering*, John Wiley, Volume 1, 1994.

Book by Zuse, Horst, Software Complexity – Measures and Methods, Document W021, *Metricating A-KINDRA BT Project 610287*, Bache, R., South Bank, 1987.

Book by Zuse, Horst, Software Complexity – Measures and Methods, IBM Res. Rep., RC7560, *System Parititioning and Its Measure*, Belady, L. A. and Evangelisti, C. J., 1979.

Book by Zuse, Horst, Software Complexity – Measures and Methods, Technical Report 149, *Evaluating Software Complexity Measures*, Weyuker, Elaine J., Courant Institute of Mathematical Sciences, New York, NY, January 1985.

Bowles, Adrian John, *Effects of Design Complexity on Software Maintenance*, Ph.D. Dissertation, Northwestern University, Evanston, IL, 1983.

Brooks, F. P. Jr.; *The Mythical Man-Month: Essays on Software Engineering;* Addison-Wesley, Reading, MA, 1975.

Channon, R. N., *On a Measure of Program Structure*; Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, November 1974.

Conte, S. D., Dunsmore, H. E., Shen, V. Y., *Software Engineering Metrics and Model*; Benjamin/Cummings Publishing Company, Menlo Park, 1986.

Cordeiro, M., *Distributed Hard Real-Time Scheduling for a Software Protoyping Environment*, Ph.D Dissertation, Naval Postgraduate School, Monterey, CA, September 2000.

IEEE Computer Society, IEEE Std. 610.12-1990 (supercedes Std. 729-1983), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, New York, NY, 1990.

Krantz, David H., and others, *Foundations of Measurement - Additive and Polynominal Representation*, Academic Press, Volume 1, 1971.

Lake, Al, “Use of Factor Analysis to develop OOP Software Complexity Metrics,” Annual Oregon Workshop on Software Metrics, Silver Falls, Oregon, March 22-24, 1992.

Luqi, Berzins, V., Yeh, R., “A Prototyping Language for Real-Time Software,” *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, October 1988.

Luqi, Shing, Mantak, “CAPS – A Tool for Real-Time System Development and Acquisition,” *Naval Research Reviews*, 1992.

Murrah, Major Michael R, *Modified Risk Model*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, 2002.

Piwowarski, Paul, “A Nesting Complexity Measure,” *Sigplan Notices*, Vol. 17, No 9, 1982.

Rombach, H. D., Bradford, T. Ulery, “Improving Software Maintenance Through Meausrement”; *Proceedings of the IEEE*, Vol. 77, No. 4, April 89.

Schach, Stephen, R., *Software Engineering*, Second Edition, IRWIN, Burr Ridge, Illinois, 1993.

Shaw, Mary, “Prospects for an Engineering Discipline of Software;” *IEEE Software*, November 1990.

Stevens, S.S., “On the Theory of Scales and Measurement,” *Science 103*, 1946,

Thayer, T. A., Liplow, M., Nelson, E.C., *Software Reliability*, North-Holland, 1978.

Tsai, W.T. and others, “An Approach Measuring Data Structure Complexity,” *COMPSAC*, 1986.

Weyuker, Elaine J., “Evaluating Software Complexity Measures,” *IEEE Transactions of Software Engineering*, Vol. 14, No. 9, September 1988.

GLOSSARY

Absolute Scale: The absolute scale is the highest scale type level in the scale typehierarchy. The admissible transformations of the absolute scale is g(x) = x. That means,no admissible transformations of the numbers x are possible [Ref. 16].

Abstraction: Is the consideration or representation of general quality or characteristics above and apart from any actual instance or specific object that prossesses that quality or characteristic [Ref. 16].

Admissible Transformation: Transforming the numbers of a statement, the truth or falsity of the statement has to be remained unchanged. For example: The statement *u(P1 o P2) = u(P1) + u(P2)* can be multiplied with *a>0* and the truth or falsity of the statement remains unchanged. We do that now: a *u(P1 o P2) = au(P1) + au(P2), for a>0* [Ref. 16].

Algebraic Difference Structures: The main concept of an algebraic difference structure is a set of objects and a quaternary operation on the set of objects *A*. A quartenary operation can be interpreted as the difference between the objects *a* and *b* is greater or equal than the difference between the objects *c* and *d* (*a ,b ,c, d, e* element of *A*). For example, this could be the preference-interval on a set of drinks (a: beer, b: wine, c: coffee and d: tea). Then *ab : cd* means that my preference to beer over wine is equal or greater than my preference to coffee over tea.

Axiom: Axioms are conditions or basic assumptions of reality. Axioms are mostly empirical, but technical ones are also possible. Axioms formulate certain empirical properties. The goal in software measurement is to figure out empirical laws about software development, software complexity, software maintainability, etc. The discovery of qualitative laws of software quality and software development is another goal of the formulation of axioms in the area of software measurement. Further goals of formulating axioms are to get a more precise terminology in the area of software measurement [Ref 16].

Binary Operation versus Concatenation Operation: Binary and concatenation operations are used as synonyms. The difference is the kind of the combination of the both objects *a, b* element of *A* to $a \circ b$, where *A* is a set of objects, like masses or flow graphs. Concatenation operations mostly mean to link two objects, for example, in a sequence [Ref 16].

Cohesion: Cohesion (alias strength) is a measure of the strength of fundamental association of processing activities (normally within a single module) [Ref 16]. Also represents modules that perform functions independently.

Complexity: The degree of 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 degree of nesting, the types of data structures, and other system characteristics. For the definition of complexity of a program or software system, we use the empirical relational systems related to a measure. Using measurement theoretic axioms a model of complexity behind a measure can be characterized [Ref. 16].

Complexity Measure: Complexity measures are a major term in the area of software measurement. However, we think it is misleading because the use of a measure depends on its empirical relation. So, the Measures of McCabe, for example, can be used as complexity measures, maintainability or testability measures. It depends on the empirical relational systems under considerations [Ref 16].

Cyclomatic Complexity: In his seminal paper McCabe [Ref. 7] derived from the cyclomatic number the term cyclomatic complexity: The overall strategy will be to measure the complexity of a program by computing the number of linearly independent paths v(G), control the "size" of programs by setting an upper limit to v(G) (instead of using just physical size), and use the cyclomatic complexity as the basis for a testing methodology [Ref 16].

Dataflow graph: A dataflow graph as opposed to flow graph or control flow graph the relationships between the data in a program [Ref 16].

Desirable Properties: Many authors formulated properties for software measures. Some authors denote these properties as desirable properties. The requirement of these properties is based on experiences, on results of experiments, on axiom systems, or on theoretical assumptions. The reason for formulating properties of software measures is to provide a standard of software measures. Many authors assume that their requirements reflect properties of reality in an acceptable way. They also use the desired properties to characterize their own proposed measures [Ref 16].

Empirical: Perceptions originating in or based on observation or experience <*empirical data*>; relying on experience or observation alone often without due regard for system and theory [Ref 8].

Empirical Conditions: Empirical conditions can be seen as an idealization of empirical facts. We call these empirical conditions, axioms as well. We use the term empirical conditions and axioms as synonyms. Furthermore, we use the terms formal and numerical conditions as synonyms. Very often, we emphasize the translation of numerical conditions to empirical conditions. The advantage of this translation is the easy interpretation of numbers. Very often, many numerical conditions only have one empirical interpretation [Ref. 16].

Empirical Law: If hypotheses of reality are validated or confirmed by many carefully designed experiments then we can call it an empirical law. For example, it is a hypothesis that adding statements increases cost of maintenance. If experiments, carefully designed, confirm the hypothesis, then it can become an empirical law. Important is to notice that empirical laws cannot be proven [Ref. 16].

Expanded File: The expanded file is located in a <<Temp>> subdirectory of the "version" directory. This file is code generated by CAPS, representing the system as a flattened hierarchy without composite operators, a flattened model of the source file/code.

Flow graph: A flow graph is a directed graph and it is the representation of the control flow of a program. It can be described by the quadruple *G=(E, N, s, t),* where *E* is the set of edges, *N* the set of nodes, *s* the start-node and *t* the exit-node with *s*, *t* element of *N*. The nodes are connected by edges.

Homomorphism: A homomorphism is a mapping from the empirical relational system to the formal relational system, which preserves all relations and structures between the considered objects [Ref. 16].

Hybrid-Measure: A hybrid-measure is a combination of two or more single measures to one measure. As a combination operator the + or * is used. However, single measures cannot be combined arbitrarily, important conditions of measurement theory have to be considered [Ref. 16].

Injective: **O**ne-to-one, mapping and injective are used as synonyms. A function *f: A* → *B,* is injective if for every $a_1, a_2 \in A: f(a_1) = f(a_2) \Rightarrow a_1 = a_2$; elements of A with the same image must be equal.

Interval Scale: An interval scale type is defined by the admissible transformation: *g(x) = ax + b, a>0.* The interval scale can be described by an algebraic difference structure (See the term algebraic difference structure). Interval scales do not play an important role in the software measurement area.

Intuitive Condition: Intuitive conditions, empirical conditions, and axioms are treated as synonyms. Empirical conditions can be falsified by observation. An example is: Program A is more difficult to maintain than Program B, can be shown to be false.

Lines-of-Code (LOC): There exist many definitions of a line of code. One of the definitions is as follows: A line-of-code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. They specifically include all lines containing program headers, declarations, and executable and non executable statements [Ref. 16].

Mapping: In many mathematical and non-mathematical situations every element of an output set is assigned to a unique object of another set (not necessarily a different set). For example, every car has a unique license plate; every U.S. citizen has a name and a unique Social Security Number. In the measurement area, a measure is defined as a mapping from empirical objects to numerical objects under the condition of a homomorphism [Ref. 16].

Measure: A measure μ is a homomorphic mapping μ: $A \rightarrow B$, where A are empirical objects, and $\rightarrow$ denotes a mapping and B is the set of real numbers [Ref. 16].

Measurement: Measurement is the process of empirical and objective assignment of numbers to the properties of objects and events in the real world in such a way to describe them [Ref. 16].

Model: The everyday meaning of the word model is defined in the dictionary in two ways. A model may be an object of imitation, such as a person who poses for artists, a role model, or some exemplar of excellence. A model may also be a representation. In this sense, a model may be a design for a new project; a template or prototype; a mold; a drawing; something that resembles something else. A model is an intentional arrangement of a portion of reality (the medium) to represent another portion of reality (the subject) such that in certain ways the model behaves like the subject; the part(s), the set(s) of details, and the abstractions of the subject that the model represents are called the viewpoint of the model; the set of ways in which the model is intended to behave like the subject is called the purpose of the model. We use models for programs written in imperative languages, software systems, object-oriented programs, etc. We also use the term qualitative model behind a measure. It is our view that behind every measure a qualitative model is hidden, which can be described by empirical conditions or axioms [Ref. 16].

Modularity: Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules [Ref.16].

Module: From our measurement view a module is a program unit that can include definitions of types, objects, and subprograms that may be accessed by other program units. In our view a module is considered to be a logical entity of significance in a software design - for example, a package, function, procedure, include file, data structure template, etc. We define a module from the software measurement perspective [Ref 16].

Nominal Scale: A nominal scale type is defined by a one-to-one transformation. Examples are license plates of cars. There is an injective relationship. For example, a car having a plate number: BNL-920 can change that plate to BNL-921 if and only if no other car with that plate exists [Ref. 16].

Ordinal Scale: An ordinal scale type is defined by the admissible transformation: strictly monotonic increasing function. It is the basis of software measurement [Ref. 16].

Ratio Scale: A ratio scale type is defined by the admissible transformation $g(x) = ax$, with $a>0$ [Ref. 16]. It allows representation in degrees of difference, such as twice as large.

Scale: Scales are defined by a homomorphism [Ref 16]. Something graduated especially when used as a measure or rule: as a series of marks or points at known intervals used to measure distances; a graduated series or scheme of rank or order <*a scale of taxation*>; a proportion between two sets of dimensions (as between those of a drawing and its original); a distinctive relative size, extent, or degree <*projects done on a large scale*>. [Ref. 8]

Scale Types: Scale types are defined by admissible transformations. For example, the ratio scale is defined: $g(x)=ax,\ a>0$ [Ref.16]. The scale types: nominal, ordinal, interval, ratio and absolute are, themselves, represented on an ordinal scale.

Software: Software comprises not just code in machine-readable form, but also all the documentation that is an intrinsic component of every project. Thus software includes the specification document, the design document, legal and accounting documents of all kinds, the software project management plan and other management documents, as well as all types of manuals [Ref. 16].

Software Engineering: The term software engineering was coined at the NATO conference in Garmisch-Partenkirchen in 1968. Since that time, there has been considerable discussion over whether software development is an engineering discipline, and the nature of software engineering itself. Mary Shaw suggests that it is not yet a true engineering discipline, but it has the potential to become one. While most of the discussion has been in academia, we have seen a steady acceptance of results by industry from the research community (e.g., formal methods, advanced design and programming languages). These results have contributed to the advances made in and the discipline of software engineering. From IEEE, software engineering is defined as: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software [Ref. 16].

Software Measure: A software measure is defined as a rule by which, a given, software related product can be quantified [Ref. 16].

Software Measurement: Software measurement is an essential component of mature software technology. It supports quality as well as project management. As far as quality management is concerned, measurement can help investigate software related phenomena and thus contribute to building better software product, process and quality models. As far as project management is concerned, measurement can help state software requirements unambiguously, assess their proper implementation throughout the software project, and achieve convincing product certification. The measurement goal of interest determines which measures are appropriate. Over the years, several 'top-down' measurement approaches for deriving measures from goals have been proposed. For example, Basili et al. define software measurement as: a technique or method that applies software measures to a (class of) software engineering object(s) to achieve a predefined goal. Such goals of measurement vary along five characteristics: what software engineering objects are being measured, why they are being measured, who is interested in these measurements, which of their properties are being measured, and in what environment they are being measured [Ref. 16].

Source Code: The written program in any programming language. We do not use a model of the program [Ref. 16]. In PSDL it is the code that can found in the source file.

Source File: The source file is where the source code can be found. The original system diagram drawn by the user generates the source file. The source file can be found in the “version” directory, which is located under the “root” directory.

Validation of a Software Measure: From IEEE, the term measure validation is defined as: The act or process of ensuring that a metric correctly predicts or assesses a quality factor [Ref. 16].

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Prof. Luqi
   Naval Postgraduate School
   Code CS/Lq
   Monterey, California

4. Prof. Valdis Berzins
   Naval Postgraduate School
   Code CS/Be
   Monterey, California

5. Prof. Man-Tak Shing
   Naval Postgraduate School
   Code CS/Sh
   Monterey, California

6. Major Michael R. Murrah
   Naval Postgraduate School
   Code CS/Mu
   Monterey, California

7. Major Joseph P. Dupont
   Ft. Leavenworth, Kansas