Software Engineering Institute
Carnegie Mellon University

FAA RESEARCH PROJECT ON SYSTEM COMPLEXITY EFFECTS ON AIRCRAFT SAFETY: CANDIDATE COMPLEXITY METRICS

William R. Nichols and Sarah Sheard

May 2015

Abstract

This special report describes the results from the second task of a two-year project to investigate the impact of system and software complexity on aircraft safety and certification for the Federal Aviation Administration. The first task focused on a literature review of what is known about complexity, definitions of complexity, and the causes and impacts of complexity. This second task focused on identifying candidate measures of complexity for systems with embedded software that relate to safety, assurability, or both.

This report begins with the task specification, motivation, and definition of complexity. The second section provides general thoughts about measuring complexity, and then the primary outcome of this research task: the candidate metrics for complexity that could relate to aircraft safety and assurance, including descriptions, sources, and strengths and weaknesses of each measure. Additional information about some of these measures is included in the appendix.

Section 3 discusses the relationship of complexity, and complexity measures, to system safety. It reports on conclusions from attempting to analyze impacts of complexity from reported accidents and then from incidents. A brief discussion of consequences of the growth of complexity follows.

Executive Summary

In this report, the second of a series, we describe the results of a task to identify a substantial number of candidate complexity metrics for the Federal Aviation Administration.

This special report describes the results from the second task of a two-year project to investigate the impact of system and software complexity on aircraft safety and certification for the Federal Aviation Administration. The first task focused on a literature review of what is known about complexity, definitions of complexity, and the causes and impacts of complexity. This second task has focused on identifying candidate measures of complexity for systems with embedded software that relate to safety, assurability, or both.

This report begins with the task specification, motivation, and the following definition of complexity:

> *Complexity is a state or quality of being composed of many intricately interconnected parts, in a manner that exceeds the ability of humans, supplemented by tools, to understand, analyze, or predict behavior.*

Section 2 provides general thoughts about measuring complexity and criteria for selection of metrics from the list of candidates. Then it presents the primary outcome of this research task: the candidate metrics for complexity that could relate to aircraft safety and assurance, as shown below. The table's 35 candidate metrics are described briefly (description, source, and strengths and weaknesses) in Section 2.2, and some are explained in more detail in the appendix.

Candidate Metric	Notes
Cyclomatic complexity (McCabe)	Well studied, internal to module
Halstead complexity	Similar to cyclomatic complexity
Henry and Kafura metrics	
Troy and Zweben metrics	Focus is on identifying problem points
Lines of code	Easy to count, tends to correlate with cyclomatic complexity
Token counts	Easy to calculate
Number of components	Easy to calculate
Function points	Available early, applicability to IMA not as clear
Number of interfaces	
Coupling	
Fan-in and fan-out	Direct, easy to count, used in many derived measures
Regularity	Hard to count
Part counting (inflows, outflows, weights)	Easy to count
Uncertainty in achieving functional requirements	No direct measures provided
Perceived complexity	No direct measures provided
Requirements complexity	Available early Depends upon subjective factors
Product complexity	Related to requirements complexity
Number of requirements	Countable early
Number of operational scenarios	Countable early
Number of critical algorithms	Countable early
Other COSYSMO cost multipliers	Must be calibrated
Average nesting	
NPath complexity	
Data flow	Graph theoretical approach requires completed code

Candidate Metric	Notes
Test coverage	Correlated with effort and ability to thoroughly verify, challenging to calculate
Number unit test cases	Correlates to test effort, necessary but not sufficient for safety
Number of safety control case test cases	Directly countable
Number of user path test cases	Directly countable
Number of requirements tests	Directly countable
Reuse	Evidence is ambiguous
Code churn	A known negative indictor, lagging
Requirements churn	A known negative indictor for quality, lagging
Development effort	Lagging indicator
Development cost	Lagging indicator
Architectural metrics	Not established or verified

Section 3 provides some initial thoughts on the relationship of complexity to system safety. A system has an architecture of many levels, with software components at the lowest level. Each element throughout the hierarchy has complexity related to the complexities of its pieces and the complexity of the interrelationships among the pieces. Identifying the complexity of a system will involve identifying many of the complexities of the layered elements and their relationships.

Section 3 continues with reporting our research findings about what causes of safety issues that are related to software can be generalized from incidents and accidents. The number of accidents with clear software contribution is miniscule; even the number of incidents is only a handful. In addition, incidents are more often related to poor expression of safety concerns during the understanding and requirements phase than to defects in code. A brief discussion of consequences of the growth of complexity follows.

1 Introduction

This special report describes the results from the second task of a two-year project to investigate the impact of system and software complexity on aircraft safety and certifiability for the Federal Aviation Administration (FAA). This task focused on identifying candidate measures of complexity that could apply to systems with embedded software and relate to safety, certification, or both. This section provides the details on this task from the work plan, the motivation for this work, and the definition of complexity being used on this project.

1.1 Task Specification

The work plan described the task as follows:

> *Identify candidate measures of complexity that apply to systems with embedded software and relate to safety, certification, or both. The task will:*

a. Identify complexity measures that have been used for systems and software, along with general rules for measuring complexity in practice (e.g., number of global variables, uses of inadequate data types, and uses of concurrency mechanisms).

b. Propose a list of common failures in the context of IMA systems (e.g., use of inappropriate design patterns) and how to cope with them (e.g., code analysis, architecture analysis).

c. Use available data sources to determine whether values of these potential measures can be known or estimated for complex systems.

d. Identify the criteria by which complexity metrics will be chosen from the list of potential metrics.

Deliverables:

White paper that lists root causes of avionics system complexity and proposes potential metrics to measure software and system complexity relevant to software system safety, including their origin, strengths and weaknesses, potential for use as metrics for the purpose of system safety and certification, and criteria for selection... [SEI 2014]

This report meets the requirements suggested above.

1.2 Motivation: Complexity and IMA Systems

Much of the work on complexity of software systems has used structural metrics at the component level. Integrated modular avionics (IMA) systems, however, are dynamic systems consisting of software combined with physical components. Scaling issues may include the number of components, heterogeneity of components, messaging and sensitivity to latency, the number of components through which messages must pass, amount of new or reused software components, total number of interfaces, and so forth.

RTCA DO-178C [RTCA 2011] is the recognized assurance standard for developing software used in airborne systems and equipment. Although use of DO-178C (and previous versions) has been successful to date, three problems are apparent:

1. Larger total inventories of software suggest more total latent defects, challenging quality levels achieved by DO-178C.
2. Larger systems containing more heterogeneous components introduce more opportunity for interaction or system defects beyond the scope addressed by DO-178C.
3. Software often absorbs the complexity of the system by collecting the data, rules, and logic that can seemingly be encoded into the software inexpensively.

The NASA study on flight software complexity [Dvorak 2009] notes that most defects in software are benign, but approximately 1% of all defects may be fatal (a conjecture consistent with other published data) and that best-in-class defect density was achieved on the Space Shuttle program with an estimated residual defect level of 0.1 per 1,000 lines of code (KLOC). The shuttle contained roughly 500 KLOC; thus, assuming a comparable error rate (an assumption for which there is no evidence; therefore this is optimistic), a rough order-of-magnitude estimate would be 50 residual defects and an expectation of <1 fatal defects. In contrast, the Joint Strike fighter contains 5.7 million lines of code

(MLOC), and the Boeing 787 contains 7 MLOC. The estimates for similarly assured software (0.1 defects/KLOC) would then be 700 residual defects with perhaps 7 fatal defects. We should not accept that DO-178C quality will remain adequate as systems continue to grow.

The second problem is that DO-178C focuses on the performance of individual components; interactions of components are beyond its scope. The experience with the Ariane 5 explosion illustrates the hazards of system effects [Lions et al. 2015]. Although a chain of failures conspired to cause the catastrophe, the root cause was that a reused component from Ariane 4 behaved inappropriately in the new environment. The problem was not defective components, but components that were not suited to work together in a specific environment (a larger and faster rocket). Not only do larger IMA systems present more opportunities for such mismatches, the economics of development will push toward reuse of components. Such reuse comes with risks.

Finally, software often becomes a "sponge" for complexity by collecting the data, rules, and logic that can seemingly be encoded into the software inexpensively. However, Stein's work with system instability suggests that complexity cannot be eliminated altogether, only moved to a different regime [Stein 2003]. For example, removing complexity in the nominal regime by automating most decisions may create significantly more complexity in the off-nominal regime, because now the operators have to react quickly, in an unfamiliar situation, when they are not used to making those decisions since that responsibility is now the system's.

1.3 Definition of Complexity

We use and evolve the concept from [Konrad 2015] that "complexity is a state that is associated with causes that produce effects." For this broad concept, a dictionary definition of "complex" will suffice (as complexity is "the quality or state of being complex") [Dictionary.com, 2015]:

> *Complex [adj.]*
>
> *1. Composed of many interconnected parts; compound; composite.*
>
> *2. Characterized by a very complicated or involved arrangement of parts, units, etc.*
>
> *3. So complicated or intricate as to be hard to understand or deal with.*

Dvorak [2009] described aspects of complexity that apply to flight software by quoting Dörner [1997]:

> *"Complexity is the label we give to the existence of many interdependent variables in a given system. The more variables and the greater their interdependence, the greater that system's complexity. Great complexity places high demands on a planner's capacities to gather information, integrate findings, and design effective actions. The links between the variables oblige us to attend to a great many features simultaneously, and that, concomitantly, makes it impossible for us to undertake only one action in a complex system. . . . A system of variables is "interrelated" if an action that affects or is meant to affect one part of the system will also affect other parts of it. Interrelatedness guarantees that an action aimed at one variable will have side effects and long-term repercussions.* [Dörner 1997, cited in Dvorak 2009]

We are using a practical definition of our own crafting for this project, in order to relate complexity to concepts of safety and assurance:

> *Complexity is a state or quality of being composed of many intricately interconnected parts, in a manner that makes it difficult for humans, supplemented by tools, to understand, analyze, or predict behavior.*

We introduce the concept of human ability as supplemented by tools because it has become clear that we use tools in order to deal with complexity. This definition allows for the fact that systems are objectively more complex every year because they have more parts, functions, purposes, and interconnections, yet any given system becomes subjectively less complex over time as we become familiar with it and able to understand, analyze, or predict its behavior.

At this point, we are looking to measure the objective aspects of complexity, those implied by "many parts," "intricate," and "interconnected." We do not address the concept of humans understanding, analyzing, or predicting behavior—or the concept of tools being used to reduce the human perception of complexity—in this report.

2 Candidate Complexity Metrics

The goal of this task was to create a list of candidate complexity metrics that would be used in future tasks to determine which metrics might be most useful. This list is shown in Table 2 in Section 2.2. Each of these metrics is then described, by description, origin, and strengths and weaknesses. Where there is additional clarifying information, an asterisk appears after the name of the metric. The appendix provides information about the candidate metrics indicated by an asterisk, as well as about the methodology for creating this list.

2.1 General Rules for Measuring Complexity in Practice

2.1.1 Measurement Goals

The goal is to determine whether a specific aircraft using avionics software can be certified as safe. Translated into language specific to the avionics system, the goal is to determine whether the system and software can be assured. This suggests that sub-goals might include the following:

1. Demonstrate through test or verification that the software components function correctly and as expected within the system. This goal implies that the verification will have a quantified completeness, provide quantified assurance of correct operation, and be completed with acceptable cost and duration.
2. Assure that the software system functions as specified. Because the correct function of components does not assure the correctness of the overall system, a quantifiable assurance of system correctness must also be possible with acceptable cost and duration.
3. Assure that the software system is correctly specified. We want to know if the system specification can be made sufficiently complete to assure safe operation.

We are concerned with complexity insofar as that complexity either (1) causes us to be uncertain about achieving these goals or (2) requires excessive cost to reduce the uncertainty to an acceptable level. The problem can be thus separated into three distinct parts:

1. measuring complexity in some objective way
2. estimating the consequences of various sources of complexity
1. estimating the effectiveness of approaches that reduce complexity, contain complexity, or deal with complexity

2.1.2 Criteria for Selecting Good Metrics

Criteria for a measurement are described by the IEEE-SA Standards Board and summarized as follows [IEEE-SA Standard Board 2009, cited in Kaner 2004]:

1. ***Correlation.*** *The metric should be linearly related to the quality factor as measured by the statistical correlation between the metric and the corresponding quality factor.*
2. ***Consistency.*** *Let F be the quality factor variable and Y be the output of the metrics function, M: F->Y. M must be a monotonic function. That is, if f1 > f2 > f3, then we must obtain y1 > y2 > y3.*
3. ***Tracking.*** *For metrics function, M: F->Y. As F changes from f1 to f2 in real time, M(f) should change promptly from y1 to y2.*
4. ***Predictability.*** *For metrics function, M: F->Y. If we know the value of Y at some point in time, we should be able to predict the value of F.*
5. ***Discriminative power.*** *"A metric shall be able to discriminate between high-quality software components (e.g. high MTTF) and low-quality software components (e.g. low MTTF). The set of metric values associated with the former should be significantly higher (or lower) than those associated with the latter."*
6. ***Reliability.*** *"A metric shall demonstrate the correlation, tracking, consistency, predictability, and discriminative power properties for at least P% of the application of the metric."*

In addition to these characteristics, we propose additional criteria to make the final selection of measures of complexity from those created during this task. Table 1 shows these additional criteria. Not all are required criteria, rather they will be used to distinguish between better and less-good metrics.

Table 1: Desirable Characteristics for Metrics

ID	Criterion
1	Objective
2	Mathematically rigorous (satisfies criteria for measurement)
3	Obtainable directly or through a proxy
4	Already available or available with minimal cost
5	Related to the effort to integrate the design
6	Related to the effort to test the design

ID	Criterion
7	Related to the effort to comprehend the design
8	Related to the effort to comprehend system behavior
9	Helpful in generating the test plan
10	Attainable; must avoid extensive or expensive data collection
11	Automatable
12	Timely; describe current status or useful for predictions rather than lagging
13	Allow for comparisons of alternatives
14	Directly countable or measureable

2.2 Metrics

Various different metrics have been used as software has grown in size and in breadth of application. This section describes some of these metrics and how they have been collected and applied. Table 2 lists the metrics.

Table 2: Candidate Complexity Metrics

Candidate Metric	Notes
Cyclomatic complexity (McCabe)*	Well studied, internal to module
Halstead complexity*	Similar to cyclomatic complexity
Henry and Kafura metrics*	
Troy and Zweben metrics*	Focus is on **identifying problem points**
Lines of code*	Easy to count, tends to correlate with cyclomatic complexity
Token counts*	Easy to calculate
Number of components*	Easy to calculate
Function points*	Available early, applicability to IMA not as clear
Number of interfaces*	
Coupling*	
Fan-in and fan-out*	Direct, easy to count, **used in many derived measures**
Regularity	Hard to count
Part counting (inflows, outflows, weights)	Easy to count
Uncertainty in achieving functional requirements	No direct measures provided
Perceived complexity*	No direct measures provided
Requirements complexity*	Available early Depends upon subjective factors
Product complexity*	Related to requirements complexity
Number of requirements*	Countable early
Number of operational scenarios*	Countable early

Candidate Metric	Notes
Number of critical algorithms*	Countable early
Other COSYSMO cost multipliers*	Must be calibrated
Average nesting	
NPath complexity	
Data flow*	Graph theoretical approach requires completed code
Test coverage*	Correlated with effort and ability to thoroughly verify, challenging to calculate
Number unit test cases	Correlates to test effort, necessary but not sufficient for safety
Number of safety control case test cases	Directly countable
Number of user path test cases	Directly countable
Number of requirements tests	Directly countable
Reuse*	Evidence is ambiguous
Code churn*	A known negative indictor, lagging
Requirements churn*	A known negative indictor for quality, lagging
Development effort	Lagging indicator
Development cost	Lagging indicator
Architectural metrics*	Not established or verified

Note that there is duplication and overlap among the metrics shown in Table 2. Many are highly derived values; for example, Troy and Zweben metrics combine numbers of modules, average number of interconnections, and fan in/fan out, adding other direct measures. Because these are candidate metrics from which ultimate metrics may be chosen, it was not appropriate at this time to reduce the overlap. The reader is referred to the appendix for more elaboration about some of the measures.

Each description includes the measure name, formula or other description, strengths and weaknesses. All in this section are considered to have potential for use as metrics for the purpose of system safety and assurance.

An asterisk indicates that the measure is discussed in more detail in the appendix.

Cyclomatic complexity (McCabe)*

Description: The number of linearly independent paths through a program module; strong indicator of testing effort.’’’

Origin: [Dvorak 2009]

Strengths and weaknesses: Advantages are that it is widely used, straightforward to compute consistently and that it correlates with effort and defect proneness. Disadvantages include that it is only for one module or program, it is only available after implementation and, because it correlates with other measures, it may not be orthogonal to them.

Halstead complexity*

Description: A kind of algorithmic complexity, measured by counting operators and operands; a measure of maintainability.

Origin: [Dvorak 2009], [Halstead 1977]

Strengths and weaknesses: Similar to Cyclomatic complexity.

Henry and Kafura metrics*

Description: Coupling between modules (parameters, global variables, calls) using graph theory, lines of code, number of procedures, fan-in, fan-out, direct coupling, indirect coupling, and other measures of information flow. An example of one of the metrics is length*(fan-in*fan-out)^2, where length is the source lines of code.

Origin: [Dvorak 2009], [Henry and Kafura 1981]

Strengths and weaknesses: Advantages include a solid theoretical basis and a number of creative potential metrics. The use of global variables in derived measures of coupling is supported by the now common practice of checking for globals statically. Disadvantages include a lack of historical data on such measures.

Troy and Zweben metrics*

Description: Modularity or coupling; complexity of structure (maximum depth of structure chart); calls-to and called-by. Include size, cohesion, and coupling. Their inputs require graphs of the modules and connections, design documents, fan-in, fan-out, average number of interconnections, number of component effects, and possible return values, among others

Origin: [Dvorak 2009], [Troy and Zweben 1981].

Strengths and weaknesses: Focus is on identifying problem points. These metrics have been used with some success in the past on programs of relatively small size, but we have found no recent citations for larger modern systems.

Lines of code*

Description: Size measure based on counting source lines of code.

Origin: Historical

Strengths and weaknesses: Lines of code tends to correlate within a domain with Cyclomatic complexity of a module.

As with other size measures, advantages include that the measure is easy to understand, generally fast to count, independent of program language, and widely applicable. Using size measures does not require deep analysis of a program's logic structure, and the industry is familiar with these types of measures. A disadvantage is that size alone does not necessarily capture the complexity from the

control flow or data flow, the strength of interface interactions that lead to emergent behavior, or the number of test cases required to provide an accurate measurement.

Token counts*

Description: Size measure

Origin: [Halstead 1977]

Strengths and weaknesses: (see Lines of Code and Halstead)

Number of components*

Description: Size measure, based on counting components

Origin: Historic

Strengths and weaknesses: Unlike lines of code, this helps to measure the system decomposition. It is easy to count, but fails to account for complexity of component interactions. This seems more promising when combined with other metrics such as fan-in and fan-out and is often a component of derived metrics such as those of Henry and Kafura [1981].

Function points*

Description: Function points counts use a structured approach to estimate software size early in the development. Counts typically use external elements, external inquiry, internal logical files, and external file interfaces. Weights, including subjective measures of complexity, can be applied.

Origin: [Albrecht 1979]

Strengths and weaknesses: (see Lines of Code). Function points are available early, prior to code. Traditionally they have been used on data processing systems, and usefulness for IMA is not as clear without developing adaptations to the domain.

Number of interfaces*

Description:

Origin: [Cataldo 2010]

Strengths and weaknesses: (see Lines of Code)

Coupling dependency (and other coupling metrics)*

Description: Coupling dependency metric (CDM), fan-in*fan-out, $(\text{fan-in*fan-out})^2$ and size*$(\text{fan-in*fan-out})^2$, count of modules that use resources, intramodule metrics, cyclomatic complexity, and lines of code using ordinal ranking of failures in operation. Also Data coupling, Stamp coupling, Control coupling, Common coupling, and Content coupling.

Origin: [Binkley and Schach 1997], [Briand 1996], [Kazman et al. 1996]

Strengths and weaknesses: Composite measure, plausible and not otherwise accounted for. Disadvantages are that it (and other coupling measures) are not readily available (although tools could be developed to recognize and count the incidences), and they have not been thoroughly studied, so their actual effect is not known at this time.

Fan-in and fan-out and information flow complexity*

Description: Fan-in is the number of incoming connections to a component, and fan-out is the number of outgoing connections. Information flow complexity *IFC* is $(fan\text{-}in + fan\text{-}out)^2$ and *Weighted IFC* is $(fan\text{-}in + fan\text{-}out)^2$.

Origin: [Henry and Kafura 1981]

Strengths and weaknesses: Advantages are that they include accessible measures of information flow among modules or components. Moreover, in avionic systems many related measures use some form of this metric. Disadvantages are that correlation with problematic components is not good, and [Kazman 1998] shows this is unsatisfactory for estimating architectural complexity.

Regularity

Description: A method based on chunking and matching a set of known patterns. The more patterns necessary to describe the system, the more complex the architecture.

Origin: [Kazman 1998]

Strengths and weaknesses: An advantage of a regularity approach is that it captures the incidence of known patterns. However, this method suffers from two disadvantages. First, the problems is that patterns must first be described, in order to have a catalogue of patterns from which to pick. Second, it is possible that two systems may, in combination, require fewer patterns than either of the two individual systems alone.

Part counting (inflows, outflows, weights)

Description: Complexity is $(Np * Nt * Ni)^{1/3}$ where, *Nt* is the number of types of parts; *Np* is the actual number of parts; and *Ni* is the number of interfaces of each of the parts. Alternately, $D_e = e_1$(inflows*outflows) + e_2(fan-in*fan-out) where e_1 and e_2 are weighting factors, inflows is the number of data entities passed to the module, outflows is the number of data entities passed from the module, and fan-in and fan-out are the number of superordinate and subordinate modules

Origin: [Meyer 1997], [Zage 1995], [Li 2015]

Strengths and weaknesses: The complete system is easily measureable, and the measures should also be available at design time and before implementation. On the other hand, the weighting factors are difficult to determine from first principles and often are just guessed. Also, the specific metrics overlap with other metrics suggested.

Uncertainty in achieving functional requirements

Description: Suh describes complexity as "a measure of uncertainty in achieving the specified FRs (functional requirements)."

Origin: [Suh 1999]

Strengths and weaknesses: While a measure of uncertainty could be a powerful measure of complexity, this concept does not suggest a specific measure.

Perceived complexity*

Description: Complexity as perceived by an observer

Origin: [Kinnunen 2006]

Strengths and weaknesses: Sources are not helpful on how to measure this.

Requirements complexity*

Description: Number of functional requirements plus number of non-functional requirements, with weights added

Origin: [Keshavarz et al., 2011]

Strengths and weaknesses:

Product complexity*

Description: This is a form of requirements complexity based on summing requirements weight by factors and described in more detail in the appendix.

Origin: [Keshavarz 2011]

Strengths and weaknesses: This measure requires subjective assessments but also provides an alternative approach to counting requirements in COSYSMO.

Number of requirements*

Description: The number of requirements for the system of interest at a given level of design. Requirements may be functional, performance, feature, or service-oriented.

Origin: [Valerdi 2008a]

Strengths and weaknesses: One advantage of using COSYSMO measures [Valerdi 2008a] is that some are included that are not otherwise captured, for example, the degree of understanding of the requirements and architecture. The approach also has been validated in other domains. However, COSYSMO measures carry several disadvantages. First, COSYSMO is not designed to measure complexity per se but to predict system development cost, duration, and effort. Second, COSYSMO

mixes process and product measures. Third, many of the measures can be determined only by subjective expert judgement. Some of the measures (such as stakeholder cohesion) may be relevant but are beyond the scope of this document.

Number of operational scenarios*

Description: This is a subset of requirements and design describing an external view of the system as it responds to a stimulus, typically in an end-to-end test scenario.

Origin: [Valerdi 2008a]

Strengths and weaknesses: (See Number of requirements) This helps to characterize the volume of external interaction of the system and the number of potential responses.

Number of critical algorithms*

Description: This represents the number of newly or significantly changed functions that require unique mathematical algorithms to be developed in order to achieve the system performance requirements. An example would be a brand-new discrimination algorithm being derived to identify a friend or foe function in space-based applications.

Origin: [Valerdi 2008a]

Strengths and weaknesses: (See Number of requirements)

Other COSYSMO cost multipliers*

Description: Requirements understanding, architecture understanding, level-of-service requirements, migration complexity, technology risk, documentation to match lifecycle, number and diversity of installations, platforms, or both, number of recursive levels of design, stakeholder team cohesion, personnel and team capability, process capability, multisite coordination, tool support

Origin: [Valerdi 2008]

Strengths and weaknesses: (See Number of requirements)

Average nesting

Description: A measure based on structural topology describing the mean or median depth of module call path structure.

Origin: [Conte et al. 1986]

Strengths and weaknesses: Control flow measures have several advantages for measuring complexity. As with size metrics, control flow metrics derived from the source code are fairly easy to compute. These metrics also tend to be relevant to the building of white box test cases and therefore can predict testability attributes. Disadvantages include that these measures are not available until the software has been built, or at least until the design is described in some detail. These do not account for complexity from data flow and do not distinguish different types of control flow.

NPath complexity

Description: A measure based on structural topology describing number of paths

Origin: [Nejmeh and Sibley 1988]

Strengths and weaknesses: See Average Nesting

Data flow*

Description: Data flow metrics are based on the use, dependency, and interaction of data within the program. Chung defined size related to data flow. The measure uses graph theory in its specific definition, though we have not yet identified the specific formulation used.

Origin: [Chung and Yang 1988]

Strengths and weaknesses: These are difficult or impractical to calculate without the aid of tools. With automated support, data flow can help to characterize the interactions among modules and identify a number of opportunities to introduce error, latency, race conditions, and so forth.

Test coverage*

Description: Test coverage includes the percentage of lines of code executed, the percentage of paths executed, the percentage of safety control paths tested and so forth.

Origin: [Hayhurst et al. 2001], [DO-178B Industry Group 2015]

Strengths and weaknesses: The advantage of code coverage metrics is that they correlate with the ability to test or verify the system. A disadvantage of code coverage metrics is that they are the focus of DO-178; therefore, code coverage is not new and may not provide enough additional insight. A second disadvantage is uncertainty about how much code is being covered.

Number unit test cases

Description: A count of unit test performed

Origin: Historic

Strengths and weaknesses: This measure focuses narrowly upon the internal module. Without careful design analysis, test cases are likely to be redundant or lack coverage. A measure of code or path coverage should be preferred.

Number of safety control test cases

Description: This is a subset of control cases specific to design for safety considerations, for example, that a component failure will not cause a safety-critical event.

Origin: [Armin Beer 2011]

Strengths and weaknesses:

Number of user path test cases

Description: A count of test cases for independent test paths tests. This is usually compared to the total number of requirements.

Origin: See Test Coverage

Strengths and weaknesses: See Test Coverage,

Number of requirements tests

Description: A count of requirements

Origin: See Test Coverage

Strengths and weaknesses: The measure is easily accessible. However, a test per requirement is not normally sufficient for safety critical systems because of the need to cover multiple conditions and inputs.

Reuse*

Description: Measure of the amount of code reused from previous systems.

Origin: [Jones 2009]

Strengths and weaknesses: Advantages are that this measure is somewhat straightforward to obtain from revision control systems, though it requires tracing code to its origin. Disadvantages are that

Code churn*

Description: Number of cumulative new, modified, and deleted lines of code.

Origin: [Jones 2009], [Nagappan and Ball 2005]

Strengths and weaknesses: This is easy to count using revision control systems, though automation of code generation may introduce unintended signals. The approach could be applied to design representations such as AADL. Although easy to count, the measure might be subject to manipulation.

Requirements churn*

Description: Number of cumulative new, added, and deleted requirements.

Origin: [Jones 2009]

Strengths and weaknesses: Advantages are that this is straight forward to obtain from revision control, that it is indicative of uncertainty or emerging understanding of needs, and suggests remaining uncertainty. A disadvantage is that this number may not relate closely to safety.

Development effort

Description: Effort to specify, design, and implement a system

Origin: [Jones 2009]

Strengths and weaknesses: Effort after the fact is straight forward to compute and can be estimated from financial data. A disadvantage is that effort estimation is not a mature field and may not be indicative of complexity or safety.

Development cost

Description: The total cost, usually in dollars, spent developing the system, or in some phase of development.

Origin: [Jones 2009]

Strengths and weaknesses: Cost is readily available from financial data. Cost will correlate with complexity and verification and validation effort. The strength of the correlation has not been documented, and cost may not correlate to safety.

Architectural metrics*

Description: Metrics can be created related to software architecture (patterns and styles, e.g.)

Origin: [Nord et al. 2014]

Strengths and weaknesses: If there were architectural metrics, the hope is that these would be available early and would relate directly both to complexity and to safety. However, the field of architectural measurement is not yet mature.

3 Relationship of Complexity to System Safety

This section, per the work plan, was originally going to propose a list of common failures in the context of IMA systems and how to cope with them. Per discussion with the FAA sponsor, that topic is deferred to task 3.4.

Instead, in the interest of becoming clearer about what the metrics are measuring, we are addressing the relationship of complexity, as it may be measured, to system and software safety. This section provides some initial thoughts on the subject.

3.1 Complexity Measures and System Safety

Figure 1 shows several of the factors that are important to consider regarding complex systems and safety. At the lowest level are software components, which are small and cohesive programs and which can be expressed in terms of inputs (upper dark box), outputs (lower dark box), and the transformation that the component performs in between them. Component complexity has been measured

for decades, with the method of McCabe [1976] evolving into a National Institute of Standards and Technology standard [Watson 1996].

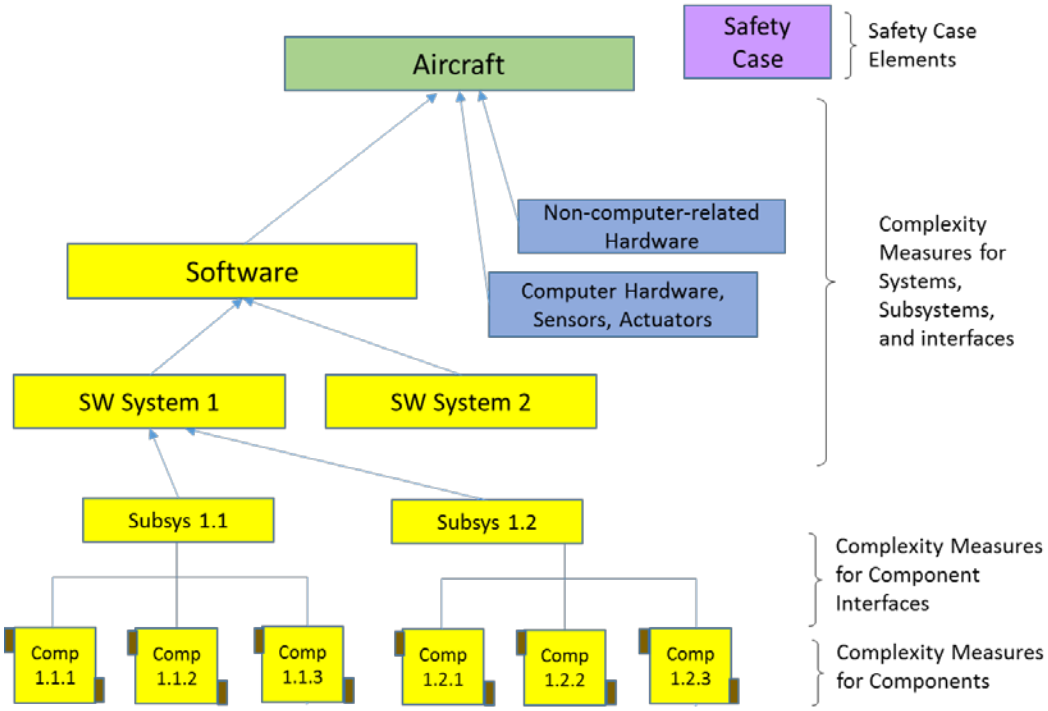

Figure 1: Elements of Safety-Critical Complex Systems

Multiple components link together to form a larger software piece (subsystem or release or other word). Both the complexity of the modules and the complexity of their interactions must be considered when evaluating the complexity of the subsystem.

A software system might have a number of these subsystems, and there may be more than one software system in a complex system. The complexity of the relationships at any level should be considered in addition to the complexities of the elements on that level.

IMA systems include this software as the logic of the Automated Controllers shown in Figure 2 [Leveson 2013]. Also included are actuators and sensors, which receive commands and provide data, respectively. These have physical reality, as do other parts of the system (e.g., the fuselage and structures connecting the actuators and the sensors). There is also a human (pilot, for an aircraft) who has access to at least some of the sensor data and can override some of the automated controller commands.

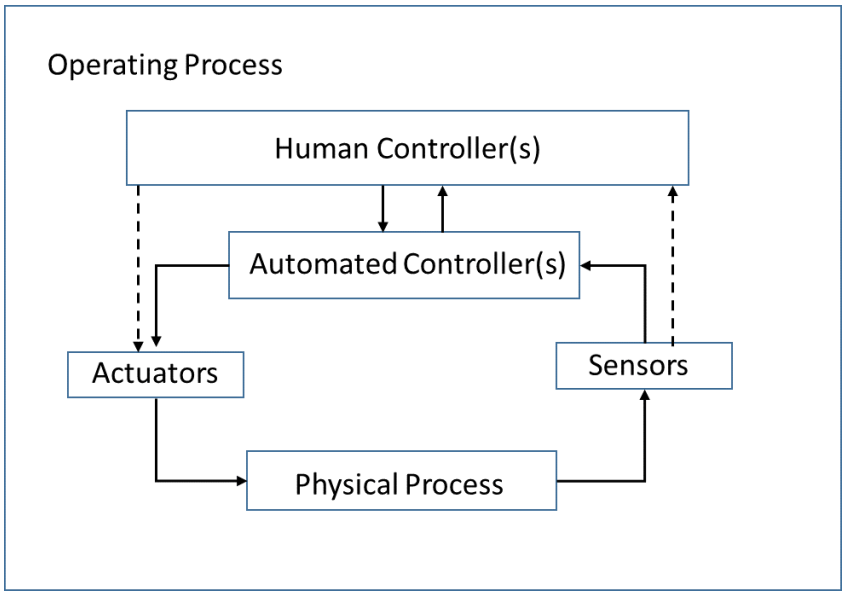

Figure 2: Control Process in Systems Theoretic Process Analysis [Leveson 2013]

Thus the complexity of the system should also address the complexity of the hardware and the interfaces between the hardware parts and the software parts, as well as among hardware parts, and between all of these and the pilot. For aircraft design purposes, complexity between the external environment, namely air and ground, and the aircraft is usually not considered in the same breath as complexity of aircraft hardware and software, possibly because these interfaces don't differ much from year to year or between proposed designs.

At the system level, inputs to the largest box in Figure 2 include operating assumptions, operating procedures, replacements to code and to hardware, and work instructions. Only if the system as a whole is safe can the system be certified. This is one reason the FAA cannot assure software by itself, only as implemented in a specific system.

3.1.1 Internal Component Complexity

Several of the metrics in Table 2 may become measures of internal component complexity. DO-178C focuses on the internal complexity and testability of software components [RTCA 2012]. This section summarizes some of the considerations that can make software components complex.

One aspect of complexity of software components is the number of independent paths through the source code. McCabe complexity [McCabe and Butler 1989] is used to estimate the number of paths and to perform basis path testing [Watson 1996]. A more complex program, therefore, will require more test cases to cover all program branches (including line of code coverage plus each path within the control structures) and be more difficult to test.

Requiring test of not only each condition (the atomic Boolean expression that returns a value of true or false) but also each decision (a composition of conditions and Boolean operators) results in condition/decision coverage. The more stringent requirement to independently test each condition that affects a decision is known as modified condition decision coverage (MC/DC).

3.1.2 System Design Complexity

We are also interested in identifying the operational properties that we might later measure. The complexity of the system design has many aspects in common with McCabe complexity (Section 3.3), including the following:

- Component internal complexity
  - Algorithmic
  - Control structure
  - Numeric
  - Data structure
  - Operations at input/output level
- Component structural complexity
- Dynamic behavior (operational behavior)
  - Operational modes/state machine transitions
  - Number and variability of policies or rules governing system behavior; non-holonomic constraints
- Configuration
  - Number of normal system configurations
  - Off-design configurations
- Functional coupling complexity (how system components interact to achieve requirements)
  - Coupling to other software components
  - Coupling to hardware components
  - Coupling between subsystems
  - Data flow complexity
  - Emergent behaviors
  - Feedback loops
  - Nonlinearity
  - Instability
- Representational Complexity

3.2 Root Causes of Avionics System Complexity

This section addresses the issue of root causes of avionics system complexity, particularly as they relate to safety. We intend to continue working on these concepts during the next task.

Causes of complexity were laboriously investigated in [Konrad and Sheard 2015]. The next big question is "In what ways does complexity cause safety to be reduced?" We explored the nature of software failures under current development practice.

3.2.1 Relevant Standards

Some relevant regulatory and guidance documents for IMA systems include DO-178C/ED-12C [RTCA 2012], its precursor DO-178B/ED-12B, and ARP4754A/ED-79A. DO-178C updates DO-178B but retains the focus on IMA components. Requirements validation is in the scope of ARP4754A/ED-79A. Additional documents provide guidance for formal methods of IMA development, software tool qualification, and object-oriented technology [RTCA 2011a, 2011b, 2011c].

3.2.2 Incidents and Accidents

We conducted a preliminary search for aircraft accidents and incidents in which software failure was a cause. The purpose of this search was to identify trends or weaknesses in current approaches. We found no hull-loss accidents that could be attributed to software developed to DO-178B or DO-178C.[1] We therefore expanded our inquiry to include incidents and accidents in which such software was a contributing factor [Daniels 2011].

Ladkin compiled a compendium of computer-related incidents in commercial software [Ladkin 2011]. From this, Daniels culled a list of incidents involving software developed to DO-178B/ ED-12B [Daniels 2011], including the following (original British spellings retained):

> *"1. TAM A320 runway overrun accident, Sao Paolo Congonhas airport, Brazil, July 2007. A contributing factor was that a software interlock might have prevented the ground spoilers from deploying.*
>
> *2. B777, anomalous flight behaviour and partial loss of Control, off Perth, W. Australia, August 2005. This was caused by a software fault in the Air Data Inertial Reference Unit (ADIRU).*
>
> *3. A320, runway overrun on landing, Taipei-Sungshan airport, Taiwan, October 2004. Thrust lever left near the Climb detent.*
>
> *4. Unknown FBW aircraft type, Byzantine failures in Flight Control System (FCS), no date.*
>
> *5. A320, braking problem on touchdown. Cardiff, August 2003. The AAIB report stated that the initiating factor in this incident was the behaviour of the Brake and Steering Control Unit (BSCU).*
>
> *6. Singapore Airlines, B747-400, Primary Flight Display information loss, January 2003.*
>
> *7. B717, in-flight engine shutdown due to electronic fault, Launceston, Australia, October 2002.*

---

1 On May 9, 2015, an Airbus NV military transport plane crashed near Seville in Spain, killing four of the six people on board, during a test flight. Whether software was a cause has not been determined, although as of May 20, investigators thought that the electronic control units may have been at fault, either in design or in installation [Agence France-Presse 2015, Kelion 2015].

8. Tu154m and B757, Midair Collision, Überlingen, Lake Constance, Germany, July 2002. Professor Ladkin claims that TCAS was a causal factor.

9. B717, dual electronic Flight Management System failure, nr. Launceston, Australia, December 2001.

10. A330, landing difficulties, Melbourne, Australia, August 2001.

11. Comair, Embraer Brasilia turboprop, Primary Flight Display information loss, 19 March 2001.

12. Iberia A320, hard landing, Bilbao, Spain, February 2001.

13. Various aircraft; airprox caused by TCAS RA manoeuvring, Trasadingen, September 2000.

14. Air UK Leisure A320 braking problem and runway overrun, Ibiza, May 1998.

15. Philippine Airlines A320 braking problem and runway overrun, Bacolod, Philippines, March 1998. Thrust lever not at idle." [Daniels 2011]

No similar compendium has yet been prepared for DO-178C. Additional incidents were associated with software developed under the significantly different standards DO-178/ED-12 or DO-178A/ED-12A. Also, an Airbus 330 crashed prior to certification on that version of the A330.

3.3 Analysis of Incidents

The majority of incidents were attributed not to software defects per se, but to requirements defects.

> *In the majority of these cases, the author's understanding is that the incident occurred, not because the software failed to meet its requirements, but because the requirements did not specify safe behaviour in all circumstances. This is not to say that airborne software developed to DO-178B/ED-12B is defect-free (this is far from being the case). Nevertheless, the in-service incidents suggest that aircraft safety could best be improved by an increased focus on requirements validation (determining that the requirements are the right requirements and that they are complete), and particularly on the interaction between flight crew and software-intensive systems.* [Daniels 2011]

From this work we can infer the following:

1. Operational loss of aircraft hull are thankfully rare, so we must increase the investigation sample by including non-loss incidents.
2. Greater numbers for statistical analysis will require expanding the search to test and development.
3. Operational data for DO-178B is limited, and data for incidents from 178C is not yet available.
4. Despite the presence of defects in the software, incidents currently appear to be dominated by requirements gaps rather than software implementation or design defects.
5. Certification separately addresses modules and system behavior.

System issues predominate the incidents that have occurred by today; however, complexity of components should not be ignored. As Figure 3 illustrates, software doubles in size approximately every two years [Feiler et al. 2013], while component defect density is more or less constant. Moreover, much of

the new code will be used to integrate components or systems. While mere integration may cause emergent behavior, the increased size implies more opportunities for operational defects. Certification will be challenged by increasing the scale of either module size or the number and heterogeneity of components. This document will, therefore, address complexity metrics at both the module and system levels.

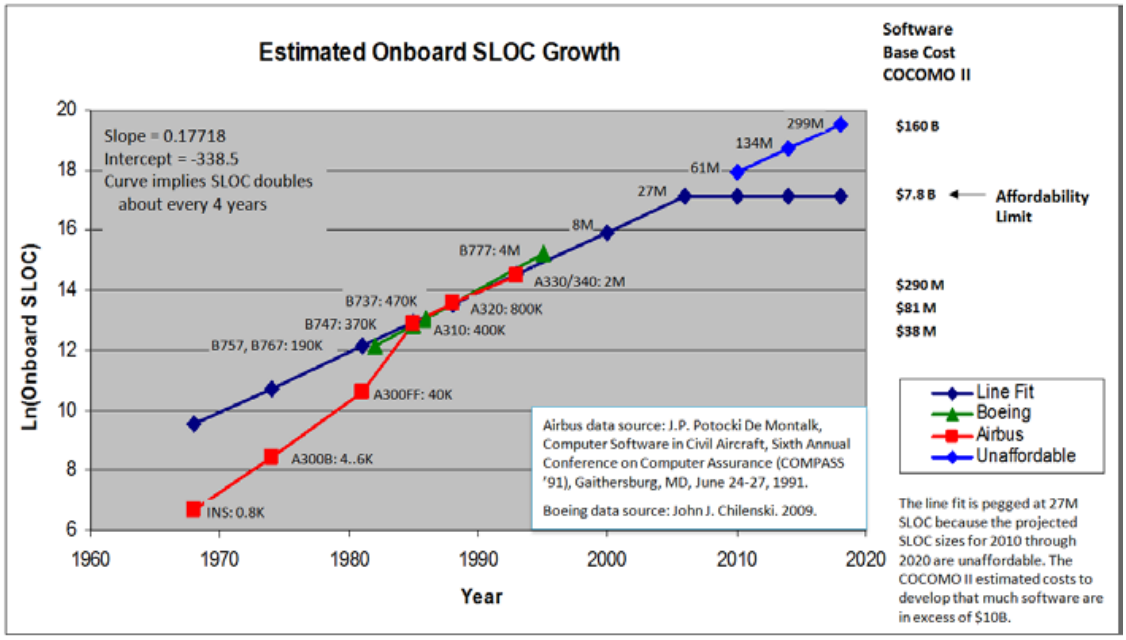

Figure 3: Increasing Size of Onboard Avionic Software [Feiler et al. 2013]

3.4 Complexity Growth

The current practice appears to have been effective—so far. Nonetheless, the increasing reliance upon software and the increasing sophistication of software systems suggest concerns for future certification as component size, number of components, and scope of software responsibility grow. These concerns include the following:

1. Increasing size of components challenges the assurance of individual components.
2. Individual components that behave correctly may still receive and process bad data, thus causing the system to malfunction.
3. Excessive component CPU time could lead to synchronization or race conditions.
4. Increasing numbers of components increases the overall complexity of the system, thus making system behavior more difficult to understand and certify.
5. The increasing scope of software use increases system complexity by expanding the number of requirements and external interfaces.

Therefore, the concern is that the scale and scope of future IMA systems may exceed the capacity to use the currently accepted certification approach. Unless alternatives are found, the systems may not be certifiable as safe.

While software has not yet been blamed for causing catastrophic accidents, there are reasons to suggest that this track record might not persist in the future. A number of incidents have occurred, many of which were not foreseen, in part due to the complexity of requirements. The ability to assure software and be compromised by ever-increasing system size and complexity.

3.5 Summary

This section described some preliminary thoughts about the relationship of a system's composition to complexity metrics and the relationship of both to safety. This section also provided more information about internal complexity of components and system complexity, which includes interface relationship complexity. It then offered some analysis of incidents in which software failure was a cause and some considerations of how the growth of complexity will affect software assurance.

4 Conclusion and Next Steps

This report reviewed the metrics that may be used to evaluate the complexity of aircraft systems that include software and computer hardware, such as IMA systems. The list, shown in Table 2, is intentionally broad but has been filtered to some extent by scope of this project (subjective and user-interface considerations are out of scope) and by maturity of the measure (we have not created new architectural metrics when a 2014 conference was unable to, for example). These metrics will be inputs into our next task, "Task 3.4, Identify the Impact of Complexity on Safety," and later tasks.

The work plan describes Task 3.4 as follows:

> *Identify the impact of complexity on aircraft certification, V&V, and flight safety margins, including reductions in margin occurring because complex system V&V is more problematic. Through interaction with FAA technical staff members, determine top-level impacts that complexity has on issues of concern to FAA.*
>
> ***Deliverables***
>
> *A draft working paper that begins to prioritize issues of certification, V&V, and flight safety as relates to the problems that are caused by complexity.* [SEI 2014]

This draft report is due by August 1, but the work of Task 3.4 will continue through the rest of FY2015. A final working paper is not a deliverable. The information created in Task 3.4 will help in creating the deliverables for Tasks 3.5 and 3.6, namely,

> *White paper containing final selection of complexity and safety measures as well as analysis and quantified contributions of the various types of complexity to system safety* [SEI 2014]

and

White paper reporting the relevance of selected metrics and demonstrating the applicability of our methods to manage complexity [SEI 2014]

which are due March 1, 2016, and July 1, 2016, respectively.

Appendix A Details: Candidate Complexity Metrics

Here we capture some of the thought processes that went into analyzing and selecting the metrics listed in Table 2. The first discussion is of metrics have been applied in avionics, and second discussion is of metrics that have been associated with software complexity.

A.1 NASA study on Avionics Software Complexity

A NASA study on flight software complexity [Dvorak 2009] explicitly addresses complexity in its Chapter 4. Table 3 lists the metrics identified in the NASA report. The same table appears in an earlier SEI handbook [Bray et al. 1997], indicating that these metrics have been used for some time.

Table 3: Complexity Metrics from NASA Study of Flight Software (FSW)

Complexity Metric	Primary Measure
Cyclomatic complexity (McCabe)	Soundness and confidence; measures the number of linearly independent paths through a program module; strong indicator of testing effort
Halstead complexity	Algorithmic complexity, measured by counting operators and operands; a measure of maintainability
Henry and Kafura metrics	Coupling between modules (parameters, global variables, calls)
Bowles metrics	Module and system complexity; coupling via parameters and global variables
Troy and Zweben metrics	Modularity or coupling; complexity of structure (maximum depth of structure chart); calls-to and called-by
Ligier metrics	Modularity of the structure chart

In the NASA study, Dvorak discusses tight coupling, complex interactions (Section 4.1.3), and interdependence:

Dvorak lists the characteristics of complexity as follows [Dvorak 2009, p. 37]:

- *How difficult [is it] for a programmer to implement the requirements the code must satisfy?*
- *How difficult [is it] for a tester to verify that the code satisfies the requirements and operates in an error-free fashion?*
- *How difficult [is it] for a lead developer to manage the development of the FSW within cost and schedule?*
- *How difficult [is it] for a FSW maintenance programmer to understand the original programmer's work if the software must be modified after launch?*
- *How difficult [is it] for a new programmer on a later mission to adapt the original FSW as heritage for the new mission?*

- *From a risk standpoint, how difficult [is it] to predict the FSW's behavior, which in turn can drive much more extensive testing and more operational "hand-holding" along with their associated higher labor costs?*

Dvorak recommends the following ways to address complexity [Dvorak 2009]:

1. *Enforce effective software requirements development and management practices.*
2. *Institutionalize the integration and participation of software engineering in all system activities.*
3. *Establish a culture of quantitative planning and management.*
4. *Collaborate on approaches to attract, develop, and retain qualified talent to meet current and future needs in government and industry.*
5. *Develop guidance and training to improve effectiveness in ensuring product quality across the life cycle.*
6. *Develop approaches, standards, and tools addressing system assurance issues throughout the acquisition life cycle and supply chain.*
7. *Improve and expand guidelines for addressing total life cycle COTS/NDI issues.*

Concerns with identifying and measuring complexity are not new, and previous attempts have been made to identify and measure the causes. In the remainder of this section, we discuss more specific measures. An advantage of these metrics is that they have been successfully used to identify problematic components. A potential disadvantage is that it is unknown whether comparing complexity components and software systems will bear fruit more broadly.

A.2 Product Metrics – Size

Size-based metrics characterize the size of the software in a way that may correlate with complexity, at least within a domain, and not necessarily linearly. Specific measures of size include

- lines of code (LOC)
- token counts (TC)
- Halstead software science (HSS) [Halstead 1977]
- number of components
- function points [Albrecht 1979]
- number of interfaces

The advantages of size measures include that they are easy to understand, generally fast to count, independent of program language, and widely applicable. Using size measures does not require deep analysis of a program's logic structure, and the industry is familiar with these types of measures. A disadvantage is that size alone does not necessarily capture the complexity from the control flow or data flow, the strength of interface interactions that lead to emergent behavior, or the number of test cases required to provide an accurate measurement.

A.3 Algorithmic Complexity

Kinnunen [Kinnunen 2006] defines algorithmic complexity as follows:

> *Given a system* S *and a coding* c *:* S → L *of system* S *in a language* L*, we call* cL*(*S*) the model of system* S *in* L*. Given a Turing machine* T*, the complexity of* cL*(*S*),* KT*(*cL*(*S*)), is defined by min{*l*(*p*) :* T*(*p*) =* cL*(*S*)}.*

The advantage to algorithmic complexity is that it appears to cover a distinct aspect. The disadvantage is that algorithmic complexity may not relevant to IMA systems.

A.4 Halstead

Halstead described some metrics of system complexity that include the following elements:

- $\dot{\eta}_1$ = the number of distinct operators
- $\dot{\eta}_2$ = the number of distinct operands
- N_1 = the total number of operators
- N_2 = the total number of operands

From these elements, one can calculate that

- the program vocabulary is $\dot{\eta} = \dot{\eta}_1 + \dot{\eta}_2$
- the program length is $N = N_1 + N_2$
- the volume is $V = N * \log_2 \dot{\eta}$
- the difficulty is $D = (\dot{\eta}_1 / 2) * (N_2 / \dot{\eta}_2)$
- the effort $E = D * V$

These metrics were correlated to effort required to produce the program and defects delivered. Advantages include a history of usage and correlation to something important to a project manager. Disadvantages include a lack of history of usage with respect to software assurability.

A.5 Cyclomatic Complexity

McCabe [McCabe 1989] defines cyclomatic complexity, the number of independent paths through a program, using graph theory as $k = e - n + 2*p$, where e is the number of edges, n is the number of nodes, and p is the number of distinct connected components in the control flow. The complexity of the design is the sum of complexity of the component modules.

Advantages of cyclomatic complexity are that it is widely used, straightforward to compute consistently and that it correlates with effort and defect proneness. Disadvantages include that it is only for one module or program, it is only available after implementation and, because it correlates with other measures, it may not be orthogonal to them.

A.6 Coupling

Kazman [Kazman 1996] addressed coupling and cohesion as an influence on testability. For example, one design tactic is to limit structural complexity of the system to avoid cyclic dependencies, encapsulate dependencies on the external environment, and reduce dependencies among components in general.

Briand and colleagues [Briand 1996] describe some structural measures of coupling theoretically, but they do not provide specific usable measures.

Binkley [Binkley and Schach 1997] examined a coupling dependency metric (CDM), fan-in*fan-out, (fan-in*fan-out)^2 and size*(fan-in*fan-out)^2, count of modules that use resources, intramodule metrics, cyclomatic complexity, and lines of code using ordinal ranking of failures in operation. Figure 4 shows the results.

Metric	Spearman Rank Correlation Coefficient	P-value
Coupling Dependency Metric	0.331	< 0.0000000363
Ordinal Scale Coupling	0.301	< 0.000000581
(Fan In) x (Fan Out)	0.298	< 0.000000747
((Fan In) x (Fan Out))2	0.298	< 0.000000747
SS x ((Fan In) x (Fan Out))2	0.287	< 0.00000177
Count of Modules that Use Resource	0.260	< 0.0000151
Accessed External Data Elements	0.259	< 0.0000156
Cyclomatic Complexity	0.043	< 0.249
Lines of Code	0.025	< 0.345

Figure 4: Coupling Metrics [Binkley and Schach 1997]

Schach identified five forms of coupling:

1. ***Data coupling*** *exists when some construct C calls another construct A and passes it one or more parameters, and each parameter is either a simple data type or a data structure all of whose elements are used by A.*
2. ***Stamp coupling*** *exists when some construct C calls another construct A and passes it a data structure as a parameter, and during the course of execution of A, only a portion of the data structure is accessed.*
3. ***Control coupling*** *exists when some construct C sends another construct A some type of control information which significantly alters the behavior of A.*

4. ***Common coupling*** *exists when some construct C and another construct A have write access to some shared common area in memory.*
5. ***Content coupling*** *exists when some construct C can access the internal address space of another construct A and make changes to A's code or data segment.*

The advantages of the Binkley couplings are that they are easy to understand and compute. The forms identified in the list by Schach have the benefit of being plausible and not otherwise accounted for. The disadvantage of the couplings in the list is that they are not readily available, although tools could be developed to recognize and count the incidences. They suffer from the further disadvantage of not having been thoroughly studied, so despite plausibility, their actual effect is not known at this time.

A.7 Fan-in and Fan-out

Henry described the information flows using fan-in and fan-out from components [Henry and Kafura 1981]. Fan-in is the number of incoming connections to a component, and fan-out is the number of outgoing connections. Derived values include the minimum, maximum, average, and sum of the products (fan-in*fan-out). Henry computed information flow complexity $IFC = (fan\text{-}in + fan\text{-}out)^2$ and $Weighted\ IFC = (fan\text{-}in + fan\text{-}out)^2$.

According to Kazman [Kazman 1998], this measure is unsatisfactory because

> *These measures do not reliably correlate with architectural complexity for two reasons. First, counter-examples abound: for instance typical utility routines and controllers have high fan-in and fan-out respectively, but may be otherwise unproblematic. Second, and more important, is that measures such as coupling/cohesion and fan-in/fan-out are not truly architectural metrics. These measure the complexity of individual parts of an architecture, but give no indication of the architecture's overall complexity.* [Kazman 1998]

An advantage of fan-in and fan-out measures is that they include accessible measures of information flow among modules or components. Moreover, in avionic systems many related metrics use some form of this one. The disadvantage is that correlation is not perfect with problematic components. Some form of composite, or average, using graph theory may be more predictive of system problems.

A.8 Actual and Perceived Complexity

Kinnunen reports [Kinnunen 2006] that Crawley[2] describes a system as composed of interrelated elements that perform a function and whose function is greater than the parts. If a system is more complex, it will be more difficult to comprehend and therefore more error prone. Crawley begins with the atomic "part" and a "module," which is a collection of elements. Parts are interconnected by one of four types: logical relational, topological, implementation, or operational. The essential complexity is that which is necessary to fulfill the functionality. Perceived complexity is the complexity as perceived by some observer. Actual complexity is the complexity within the system, which must be at least as great as the essential complexity.

---

2 Edward Crawley. *System Architecture: Course Notes*. MIT, 2005.

Perceived complexity is an aspect of complexity that is otherwise difficult to describe. The disadvantage is that the sources are not very helpful on how to measure this complexity.

A.9 Requirements Complexity

The previous sections covered metrics in which complexity was measured from product implementation. In this section, we review prior work that attempts to measure product complexity from the problem requirements. The requirements also suggest the minimum (or essential) complexity required to solve the problem. For FR = functional requirements, and NFR = of non-functional requirements, Keshavarz and colleagues define the requirements complexity, *RC*, as *FR* + *NFR*. A formulation that applied weight coefficients to requirements is as follows [Keshavarz et al. 2011]:

$$FR = \left(\sum_{i=1}^{3} Coefficient_i \times No_i \right) \times \left(\sum_{j=1}^{level-1} \sum_{k=1}^{n} SF_k \right)$$

Table 4: Volatility Types and Its Coefficient [Keshavarz et al. 2011]

Type of Volatility	Volatility Coefficient
Stable	1
Rarely	2
Often	3

In their Equation 16, Keshavarz and colleagues calculate

$$NFR = \left(\sum_{i=1}^{3} Coefficien\, t_i \times No_i \right) \qquad (16)$$

Table 5: Importance Factors [Keshavarz et al. 2011]

Importance Degree	Importance Coefficient
Optional	1
Desirable	2
Essential	3

In their Equation 18, Keshavarz and colleagues calculate product complexity as

$$PC = (IOC \times RC) + \sum Cost\ Driver\ Product\ Attributes$$

Table 6: Requirements Factor Weightings [Keshavarz et al. 2011]

Attribute	Very Low	Low	Nominal	High	Very High
Required software reliability	0.75	0.88	1.00	1.15	1.40
Size of application database		0.95	1.00	1.08	1.16
Complexity of the product	0.7	0.85	1.00	1.5	1.3

The advantage of requirements complexity is that it provides an early estimate of the essential complexity in a solution. One disadvantage is that weighing factors are subjective. A second problem is that the total complexity still depends upon the implementation complexity.

COSYSMO Factors

COSYSMO (COnstructive SYStems engineering cost MOdel) is a parametric cost estimator for systems development [Valerdi 2008]. COSYSMO was adapted to extend COCOMO ([software] COnstructive COst MOdel) from the software to a systems environment.

Specific measures of system size in COSYSMO include

- number of system requirements
- number of operational scenarios
- number of critical algorithms

COSYSMO also includes cost multipliers, which are used to scale costs based on size as the following factors increase:

- requirements understanding
- architecture understanding
- level-of-service requirements
- migration complexity – influence of legacy systems
- technology risk – maturity, readiness, and obsolescence of technology
- documentation to match lifecycle – breadth and depth of required documentation
- number and diversity of installations, platforms, or both
- number of recursive levels of design – number of levels of the work breakdown structure
- stakeholder team cohesion
- personnel and team capability
- process capability
- multisite coordination – location of stakeholders and coordination barriers
- tool support

One advantage of using COSYSMO measures is that some are included that are not otherwise captured, for example, the degree of understanding of the requirements and architecture. However, COSYSMO measures carry several disadvantages. First, COSYSMO is not designed to measure complexity per se but to predict system development cost, duration, and effort. Second, COSYSMO mixes process and product measures. Third, many of the measures can be determined only by subjective expert judgement. Some of the measures (such as stakeholder cohesion) may be relevant but are beyond the scope of this document.

Control Flow

Control flow measures of a program are based on its structural topology. Specific measures of software component (or program) control flow size include

- McCabe's cyclomatic complexity [McCabe 1976]
- average nesting level [Conte and Yang 1986]
- NPath complexity [Nejmeh and Sibley 1988]

Control flow measures have several advantages for measuring complexity. As with size metrics, control flow metrics derived from the source code are fairly easy to compute. These metrics also tend to be relevant to the building of white box test cases and therefore can predict testability attributes. Finally, within a domain, complexity tends to correlate with size [Huang and Liu 2013, Jay 2009]. Disadvantages include that these measures are not available until the software has been built, or at least until the design is described in some detail. These do not account for complexity from data flow and do not distinguish different types of control flow.

Data Flow

Data flow metrics are based on the use, dependency, and interaction of data within the program. Chung defined size related to data flow [Chung and Yang 1988]. The measure used graph theory in its specific definition, though we have not yet identified the specific formulation used.

A.10 Test Coverage

This section focuses on code coverage. Other measures of test cases include requirements, use paths, and safety control. Each of these types of test cases provide an indicator of complexity.

Complex systems are difficult to test because of a large state space and the larger number of interconnections among elements of the system. The challenges are of scale—including number of tests required, the effort required to build and evaluate test results, and time required to execute the tests—and observabilty. Issues of scale are related to structural measures of program complexity.

The DO-178 Industry Group [DO-178B Industry Group 2015] estimates that 20% of avionic software must satisfy Level C criteria, approximately 30% of avionic software must satisfy the stricter Level B criteria (including DC coverage), and approximately 40% of avionic software must satisfy the strictest Level A criteria (including MC/DC coverage). Christopher Ackerman reported [Ackermann 2010] that an industry source claimed 7 weeks of duration to complete MC/DC tests on 20,000 lines of code, which is effort-intensive, and becoming prohibitively so as code size reaches 1,000,000 lines or more. The counterpoint is that MC/DC is more effective than alternative test strategies.

A separate criticism of MC/DC is that structural choices, for example, in-lining or decomposing a module, can reduce the required coverage without effectively changing the code [Rajan et al. 2008]. The authors suggest creating a metric that is independent of code structure but do not propose a specific metric. The same authors report in a subsequent paper that rigorous coverage metrics (for example, MC/DC [Hayhurst et al. 2001]) provide better fault finding than black box alternatives [Staats et al. 2010].

Indeed, architectural decisions can affect a system's testability by limiting complexity [Bass et al. 2003] to no more than the minimum complexity required to implement the system. (Complexity beyond this minimum is sometimes referred to as *accidental* complexity.) The tactics recommended are to add controllability and observability to the system.

Potential design approaches include "hide, shrink, and organize" [Kazman and Kruchten 2012]. Unfortunately, Kazman and Kruchten's presentation provides no specific measures. Although use of organization as a tactic suggests design patterns, no general rules are yet available because the selection of specific patterns depends upon multiple factors such as previous design decisions, the weighting of competing or reinforcing quality attributes, and so forth.

The advantage of code coverage metrics is that they correlate with the ability to test or verify the system. A disadvantage of code coverage metrics is that they are the focus of DO-178; therefore, code coverage is not new and may not provide enough additional insight. The broader problem is that DO-178 does not directly account for the complexity of interacting components and for tests of paths through the entire system.

A.11 Development Process Metrics

Development process metrics are collected during development for management of the effort, for each phase of the software development lifecycle. Some key metrics include [Jones 1996]

- reuse
- code churn
- requirements churn
- defects and defect density
- programming standards violations
- cost
- direct effort
- numbers of developers
- schedule durations
- schedule trends
- developer capability
- process and regulatory compliance

Advantages of process metrics include the following:

- capture the context of the development effort
- can be used to estimate defect potentials
- assist in trade-off of test and other verification techniques
- can be obtained with an instrumented development environment
- can be useful in calibrating and using predictive models of development cost, duration, and defects (for example, see COSYSMO [Valerdi 2008])

A challenge to using process metrics is that the quality of the data depends upon the environment. Instrumentation of development environments often leaves gaps. Quality of data recorded by hand depends upon the skill of the data recorders. In addition, benefits and risks of reuse and relevance of churn in requirements have been reported [Jones 2009] but have not specifically been linked to complexity.

A.12 Architectural Metrics

The field of architectural measurement is not yet mature. It is hoped that in the future best practices will be codified and complexity will be measured. A summary from the First International Workshop on Software Architecture Metrics suggests future work to create a metrics catalogue, derive metrics from patterns and styles, establish a common test bed for architecture metrics, and develop good metrics computation tools [Nord et al. 2014].

A.13 Summary

Complexity of a system results from the number of elements it has (its size), from the diversity of its element types, and from the connectivity of the elements. The essential complexity of a system is necessary for the system to function, to perform the complex task asked of it, to address a complex problem. Accidental complexity is not strictly required by the system, but has been introduced for other reasons, and is therefore a candidate for reduction if possible. Various measures of complexity attempt to characterize aspects of the size, diversity, or connectivity. Although these aspects are understandable one by one, it is not always clear how these aspects combine. The overarching goal is assurability, which is related to verifiability or testability.

Among the metrics identified, the citation dates suggest that most have been in use for a long time. Many of these metrics are structural. Some, including the number of test cases required for MC/DC [Hayhurst et al. 2001] testing, have been criticized for producing significantly different results depending on arbitrary decomposition decisions. The arbitrary nature should be addressed if robust metrics for optimizing cohesion and coupling are to be found.

Architectural metrics, though they appear promising, have not yet matured sufficiently for us to know what metrics are reliable, much less how strongly they correlate with complexity. Development of architectural metrics is a current field of research. Architectural metrics might better characterize global structure and appropriate use of patterns and tactics.

Bibliography

Ackermann, Christopher. 2010. “MCDC_Coverage_02in a Nutshell.” Fraunhofer USA, Inc. http://www.slidefinder.net/n/nutshell_christopher_ackermann/mcdc_coverage_02/7587078.

Albrecht, Allan. 1979. “Measuring Application Development Productivity.” In *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, 83–92. Monterey California: IBM Corporation.

Armin Beer, Bernhard Peischl. 2010. “Testing of Safety-Critical Systems: A Structural Approach to Test Case Design.” In *Advances in Systems Safety*. Springer, 187–211.

Basili, Victor, Gianluigi Caldiera, and H. Dieter Rombach. “The Goal Question Metric Approach,” *Encyclopedia of Software Engineering*, Vol. 1, pp. 528–532, Wiley, 1994. https://www.cs.umd.edu/~basili/publications/technical/T89.pdf.

Bass, Len, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice*. 2nd edition. Addison-Wesley.

Binkley, Aaron B., and Stephen R. Schach. 1997. “Metrics for Predicting Run-Time Failures.” TR 97–03. Nashville, TN.

Bray, Michael, Kimberly Brune, David A. Fisher, John Foreman, Jon Gross, Gary Haines, William Mills, and Robert Rosenstein. 1997. “C4 Software Technology Reference Guide: A Prototype.” CMU/SEI-97-HB-001. Pittsburgh: Carnegie Mellon Software Engineering Institute.

Briand, L. C., S. Morasca, and V. R. Basili. 1996. “Property-Based Software Engineering Meassurement.” *IEEE Transactions on Software Engineering* 22 (1): 68–86. doi:10.1109/32.481535. http://dl.acm.org/citation.cfm?id=229713.229722.

Chung, C. M., and M. G. Yang. 1988. “A Software Maintainability Measurement.” In *Proceedings of the 1988 Science, Engineering and Technololgy*, 12–16.

Conte, S. D., H. E. Dunsmore, and V. Y. Shen. 1986. *Software Engineering Metrics and Models*. Benjamin/Cummings Publishing Company, Inc.

Daniels, D. 2011. “Thoughts from the DO-178C Committee.” In *6th IET International Conference on System Safety 2011*, C31–C31. IET. doi:10.1049/cp.2011.0266. http://digital-library.theiet.org/content/conferences/10.1049/cp.2011.0266.

Dictionary.com, 2015. (From http://dictionary.reference.com/browse/complex, retrieved 5/26/2015)“DO-178B Q & A.” 2015. Accessed April 25. http://www.do178site.com/do178b_questions.php.

DO-178 Industry Group. “DO-178B Q & A.” 2015. Accessed April 25. http://www.do178site.com/do178b_questions.php.

Dörner, Dietrich. 1997. *The Logic of Failure: Recognizing and Avoiding Error in Complex Situations*. New York, New York, USA: Basic Books. http://www.amazon.com/The-Logic-Failure-Recognizing-Situations/dp/0201479486.

Dvorak, Daniel L. 2009. “NASA Study on Flight Software Complexity.” In *AIAA Infotech@Aerospace Conference and AIAA Unmanned...Unlimited Conference*, 264 pp. American Institute of Aeronautics and Astronautics. http://oceexternal.nasa.gov/OCE_LIB/pdf/1021608main_FSWC_Final_Report.pdf\nhttp://arc.aiaa.org/doi/pdf/10.2514/6.2009-1882\nhttp://arc.aiaa.org/doi/abs/10.2514/6.2009-1882.

Feiler, Peter, John Goodenough, Arie Gurfinkel, Charles Weinstock, and Lutz Wrage. 2013. “Four Pillars for Improving the Quality of Safety-Critical Software-Reliant Systems.” Pittsburgh. http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=47791.

Halstead, Maurice H. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. New York, New York, USA: Elsevier Science, Inc. http://dl.acm.org/citation.cfm?id=540137.

Hayhurst, Kelly J., Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. 2001. “A Practical Tutorial on Modified Condition / Decision Coverage.” NASA/TM-2001-210876. Hampton, Virginia.

Henry, Sallie, and Dennis Kafura. 1981. “Software Structure Metrics Based on Information Flow.” *IEEE Transactions on Software Engineering* SE-7 (5): 510–518. doi:10.1109/TSE.1981.231113. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702877.

Huang, Fuquen, and Bin Liu. 2013. “Study on the Correlations Between Program Metrics and Defect Rate by a Controlled Experiment.” *Journal of Software Engineering* 7 (3): 114-120. doi:10.3923/jse.2013.

IEEE-SA Standard Board. 2009. “IEEE Standard for a Software Quality Metrics Methodology.” IEEE Std 1061™-1998 (R2009). Vol. 1998. New York: IEEE.

Jay, Graylin. 2009. “Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship.” *Journal of Software Engineering and Applications* 2 (3): 137–143. doi:10.4236/jsea.2009.23020.

Jones, Capers. 1996. *Applied Software Measurement: Assuring Productivity and Quality*. 2nd ed. Hightstown, NJ, USA: McGraw-Hill.

———. 2009. *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*. 1st ed. New York: McGraw-Hill Osborne Media.

Kaner, Cem, and Walter P. Bond. 2004. “Software Egineering Metrics: What Do They Measure and How Do We Know?” http://testingeducation.org/a/metrics2004.pdf.

Kazman, Rick. 1998. “Assessing Architectural Complexity.” In *Proceedings of 2nd Euromicro Working Conference on Software Maintenance And Reengineering (CSMR 98)*. IEEE Computer Society Press, 1998. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.73.2559.

Kazman, Rick, G. Abowd, L. Bass, and P. Clements. 1996. “Scenario-Based Analysis of Software Architecture.” *IEEE Software* 13 (6): 47–55. doi:10.1109/52.542294. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=542294.

Kazman, Rick, and Philippe Kruchten. 2012. “Design Approaches for Taming Complexity.” In *SysCon 2012 IEEE International Systems Conference, Proceedings*, 519–524. doi:10.1109/SysCon.2012.6189488.

Keshavarz, Ghazal. 2011. “Metric for Early Measurement of Software Complexity.” *International Journal on Computer Science and Engineering* 3 (6): 2482–2490.

Kinnunen, Matti J. 2006. “Complexity Measures for System Architecture Models.” Massachusetts Institute of Technology.

Konrad, M., and S. Sheard. 2015 “FAA Research Project: System Complexity Effects on Aircraft Safety: Literature Review Task 3.2: Literature Search to Define Complexity for Avionics Systems.” Special Report CMU/SEI-2015-SR-006. Pittsburgh: Carnegie Mellon Software Engineering Institute.

Ladkin, L. B. 2011. “Computer Related Incidents with Commercial Aircraft.” http://www.rvs.uni-bielefeld.de/publications/compendium/incidents_and_accidents/index.html.

Leveson, Nancy. 2013. An STPA Primer, Version 1, August 2013. http://sunnyday.mit.edu/STPA-Primer-v0.pdf

Li, J. J., E. Wong, D. Zage, and W. Zage. 2015. “Validation of Design Metrics on a Telecommunication Application.” Accessed April 2. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.78.930.

Lions, J. L., and the Inquiry Board. “ARIANE 5 Failure: Full Report.” 2015. Accessed May 17. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

McCabe, Thomas J. 1976. “A Complexity Measure.” *IEEE Transactions on Software Engineering* SE-2 (4): 308–320.

McCabe, Thomas J., and Charles W. Butler. 1989. “Design Complexity Measurement and Testing.” *Communications of the ACM* 32 (12): 1415–1425. doi:10.1145/76380.76382. http://portal.acm.org/citation.cfm?doid=76380.76382.

Meyer, Marc H., and Alvin P. Lehnerd. 1997. *The Power of Product Platforms*. Free Press.

Nagappan, N., and T. Ball. 2005. “Use of Relative Code Churn Measures to Predict System Defect Density.” *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*: 284–292. doi:10.1109/ICSE.2005.1553571. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1553571.

Nejmeh, Brian A., and Edgar H. Sibley. 1988. “NPATH: A Measure of Execution Path Complexity and Its Applications.” *Communications of the ACM* 31 (2): 188–200. doi:10.1145/42372.42379. http://dl.acm.org/citation.cfm?id=42372.42379.

Nord, Robert L., Ipek Ozkaya, Heiko Koziolek, and Paris Avgeriou. 2014. “Quantifying Software Architecture Quality Report on the First International Workshop on Software Architecture Metrics.” *ACM SIGSOFT Software Engineering Notes* 39 (5): 32–34. doi:10.1145/2659118.2659140. http://dl.acm.org/citation.cfm?id=2659118.2659140.

Park, Robert E., Wolfhart B. Goethert, and William A. Florac. 1996. “Goal-Driven Software Measurement: A Guidebook.” CMU/SEI-96-HB-002. Pittsburgh: Carnegie Mellon Software Engineering Institute.

Rajan, Ajitha, Michael W. Whalen, and Mats P. E. Heimdahl. 2008. “The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage.” In *Proceedings of the 13th International Conference on Software Engineering - ICSE ’08*, 161. New York, New York, USA: ACM Press. doi:10.1145/1368088.1368111. http://dl.acm.org/citation.cfm?id=1368088.1368111.

RTCA, Inc. “Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations.” DO-297. www.rtca.org.

———. 2011a. “Formal Methods Supplement to DO-178C and DO-278A.” DO-333. www.rtca.org.

———. 2011b. “Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A.” DO-332. http://www.rtca.org/store_product.asp?prodid=848

———. 2011c. “Software Tool Qualification Considerations.” DO-330. http://www.rtca.org.

———. 2012. “Software Considerations in Airborne Systems and Equipment Certification.” DO-178C/ED-12C. Washington, D.C. http://www.rtca.org/store_list.asp.

Software Engineering Institute. 2014-2016 Work Plan for the Federal Aviation Administration (FAA) PWS 5-427 A1 Version 1.0, 2014.

Staats, Matt, Michael W. Whalen, Mats P. E. Heimdahl, and Ajitha Rajan. 2010. “Coverage Metrics for Requirements-Based Testing: Evaluation of.” In *Proceedings of NFM 2010, April 13-15, 2010, Washington D.C., USA.*, 161–170.

Stein, Gunter. 2003. “Respect the Unstable: The Practical, Physical (and Sometimes Dangerous) Consequences of Control Must Be Respected, and the Underlying Principles Must Be Clearly and Well Taught.” *IEEE Control Systems Magazine* (August): 12–25.

Suh, Nam P. 1999. “Engineering Design: A Theory of Complexity, Periodicity and the Design Axioms.” *Research in Engineering Design* (11): 116–131.

Troy, Douglas A., and Stuart H. Zweben. 1981. “Measuring the Quality of Structured Designs.” *Journal of Systems and Software* 2 (2): 113–120. doi:10.1016/0164-1212(81)90031-5.

Valerdi, Ricardo. 2008. *The Constructive Systems Engineering Cost Model (COSYSMO): Quantifying the Costs of Systems Engineering Effort in Complex Systems.* Saarbrücken, Germay: VDM Verlag.

Watson, Arthur H. 1996. “NIST Special Publication 500-235: Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.” SP 500-235. http://www.mccabe.com/pdf/mccabe-nist235r.pdf.

Zage, Wayne M., Dolores M. Zage, and Cathy Wilburn. 1995. “Avoiding Metric Monsters: A Design Metrics Approach.” *Annals of Software Engineering* 1 (1): 43–55. doi:10.1007/BF02249045. http://link.springer.com/10.1007/BF02249045.

Acknowledgments

The project team members, in addition to the authors, are Mike Konrad, Charles Weinstock, and Greg Such. For this task they helped frame the discussion and review the ideas.

Maureen Brown and Dave Zubrow provided important review comments, making this report better.

Tamara Marshall-Keim provided patient, persistent, high-quality, and timely technical editing.

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu | www.cert.org
Email: info@sei.cmu.edu

This material is based upon work funded and supported by Federal Aviation Administration under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

DM-0004270