Software Engineering Institute
Carnegie Mellon University

FAA RESEARCH PROJECT ON SYSTEM COMPLEXITY EFFECTS ON AIRCRAFT SAFETY: ESTIMATING COMPLEXITY OF A SAFETY ARGUMENT

Michael Konrad, Sarah Sheard, Chuck Weinstock, and William R. Nichols

February 2016

Executive Summary

Growth of avionics complexity threatens the ability of the FAA to certify the safety of an aircraft using a new avionics set. This project aims to determine when such safety certification will become impossible because of avionics complexity. Previous reports from this project have shown that:

- The concept of complexity is broad and specific subsets must be identified that would have meaning for a specific purpose. [Konrad 2015]
- Complexity metrics for the systems and software have many possible variations, including many size and interconnectivity measures. [Nichols 2015]
- The method for determining the certifiability of an aircraft matters in discussion of whether an aircraft is safe. [Sheard 2015]

This report presents a well-developed formula for estimating the complexity of an avionics system, in a way that selects a specific kind of complexity from the many variations, and directly connects the complexity to the size of its safety argument. We measure the complexity of the safety argument. Namely, what subclaims will have to be made and how will they be structured to argue successfully that the system is safe.

It is taken as an axiom that today's aircraft, with today's avionics, are safe. We are concerned not with incremental growth in complexity from today, but rather with order-of-magnitude jumps that may occur in the next decade or two. This means that our goal is to come up with a formula for complexity and its relationship to safety that distinguishes order-of-magnitude differences and, importantly, can be estimated at an early enough point in aircraft development that action can be taken to make the system less complex if there will be a problem certifying with the initial complexity.

We have developed a means to estimate of the size of the safety case, which can be used to compare future avionics systems to today's avionics systems, and then determine how difficult it will be to assure the safety of a future aircraft compared to today's aircraft. The order of magnitude of the result can give an early indication of whether the avionics will be usable on a safety-critical aircraft.

The complexity formula is given in Section 3.3, which is not duplicated here because of the precision of definitions required to understand the formula. In essence, we study the number of ways a failure (of software or hardware) can propagate out of a component to another component. Making several

simplifying assumptions (Section 3.1), it can be shown that the effort to assure a safety claim will be proportional to this number.

We provide step-by-step instructions for applying the formula in Section 4, related to the case described in Section 7 and the exact case provided to testers, in Appendix A, including showing how one tester applied the formula to that case. We provide thoughts about automating and using this formula in Sections 5 and 6, respectively.

1 Introduction

This report presents a well-developed formula for estimating the complexity of an avionics system, in a way that selects a specific kind of complexity from the many variations, and directly connects the complexity to the size of its safety argument. We measure the complexity of the safety argument. Namely, what subclaims will have to be made and how will they be structured to argue successfully that the system is safe.

It is taken as an axiom that today's aircraft, with today's avionics, are safe. We are concerned not with incremental growth in complexity from today, but rather with order-of-magnitude jumps that may occur in the next decade or two. Our goal is to come up with a complexity formula that distinguishes order-of-magnitude differences and, importantly, can be estimated at an early enough point in aircraft development that action can be taken to make the system less complex if there will be a problem certifying with the initial complexity.

This special report describes the results of Task 3.5, the fourth deliverable in a two-year project to investigate the impact of system and software complexity on aircraft safety and certifiability for the Federal Aviation Administration (FAA). The Statement of Work described this task as follows:

> *3.5 Quantify the Effects of System Complexity on Aircraft Safety and Identify Relevant Complexity Measures*
>
> *Quantify the effect of system complexity on system safety, as possible using available data sources. Identify relevant measures of complexity for avionics systems and select measures that are of particular interest for system validation and certification. Show how the metrics relate (e.g., how the code quality impacts the system certification) to existing standards for avionics system development (e.g., DO-178C, ARP-4754A, ARP-4761A).*
>
> *Deliverables*
>
> *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.*

Through a number of conversations between the FAA and SEI since the initiation of this project, a slight change in tasking has arisen. This deliverable is focusing on the means of calculating the complexity of the system in terms of its safety case, namely, how difficult will it be to prove safety for this system. This is clearly an appropriate interpretation of complexity for a safety-critical system. Because safety cases are not discussed per se in existing avionics system standards, standards are included sparsely in this report, only as they relate to the error model and the impact of complexity on testing.

In addition, this report includes some work originally conceived of for a future task, Task 3.6, as shown.

> *3.6 Test the Identified Metrics*
>
> *Test the identified metrics on a jointly-agreed to representative avionics system to prove the applicability of the proposed approach. Analyzing an existing avionics system, using either existing source code or specifications, and highlight the complexity of the system using the selected metrics. Depending on the selected system and the accuracy of the artifacts under analysis, quantify the impact of the suggested approach*

The complexity formula has been tested by asking the two project team members who did not develop the formula to follow it for a specific example shown in Appendix A. Results of that test are given in Section 7 and in Appendix A.

2 Problem

Today's safety-critical systems continue to grow in functionality and performance. Individual components themselves also become more complex, and consequently, system integration and testing continue to grow more challenging. While incremental improvements are being made in how we develop systems and software, our system and software development, integration, and V&V capabilities may not be rising fast enough to understand, predict, and test component-to-component interactions that can give rise to unexpected hazards and accidents.

With the increase in demand for and availability of new functionality and for the ability to operate in environments of greater interconnectedness, system designs are becoming increasingly complex, as are the developer's development, integration, and test strategies. Therefore, the set of development artifacts that are available for review for certification purposes are likewise getting more complex. While improved design, analysis, and test capabilities are mitigating some of the challenge of reviewing these development artifacts, some of these improvements add to the overall complexity of the certification task.

The goal of this project task is to provide a way to estimate the size of the safety review and assurance task. Actually completing a full safety certification review takes a great deal of work over a period of years and requires access to design details that are not available early in the system development. Thus, the full review would not be useful for early predictions. However, it is possible to *estimate* how

difficult the task will be, given an error model (also called a fault model) and a system architecture that identifies the functions of a top level of components. If the certification task is known to be too large, then different actions may be taken to reduce the complexity, such as prototyping a new design or performing early model-based safety reviews.

We have focused on how to estimate the size of a certification review early.

2.1 Approach to a Solution

Toward identifying an approach to assessing the size of a safety review/assurance task, we identified what information was essential to have in conducting such a task. To identify such information, we reviewed prior work at the SEI [Feiler 2015] employing AADL and Confidence Maps (a framework for developing assurance cases [Goodenough 2013]) in developing an assurance case for a stepper motor subsystem. An observation made in that technical report [Feiler 2015] seemed key: the top-level cases to consider in a safety argument trace to the failure conditions that can propagate out of one system component to affect some other component. With thus observation, we decided to approach the problem of providing an estimate of the safety review/assurance task by making these simplifying assumptions:

- A reasonably complete, high-level design (say, down to some level or tier of subcomponents) has been developed
- A hazard analysis has been performed on this design, resulting in a fault model identifying the various sources of hazards and how they might propagate
- The safety review task requires reasoning about how the system will behave when a failure condition propagates from one component to a second
- The evaluation and disposition of all such cases (determining whether the case is resolved or whether a further question needs to be asked) take (about) equal time

Then under these simplifying assumptions, the length of the review task should be proportional to a count of all cases in which a failure condition propagates from a first component to a second. This count is, of course, a very crude approximation to the time taken, but employed over a range of different cases, a constant of proportionality can be determined that should at least correlate with the actual time needed. In any case, absent two elements, some kind of count of failure conditions and the different ways they can escape to reach other components are a good first guess. Those two elements that would improve the estimate of the time for the certification task, but are not generally available early are:

1. A cognitive model of a reviewer's ability to review the correctness of ("cognitively digest") an argument consisting of some number of conditions with so many variables and also instances of causal inference
2. A characterization of how the system's responses to escaped failure conditions maps into that cognitive model.

3 Complexity Formula

In this section, we discuss how to estimate the potential size of an argument about system safety, also known as the "Complexity Formula."

Section 3.1 covers the main assumptions and inputs. Section 3.2 covers the terminology used. Section 3.3 explains the notation used in the formula and the presents the formula. Section 3.4 provides the rationale for the formula. Section 3.5 addresses generalizations and limitations of the formula, and addresses several other points. Section 3.6 addresses a necessary preparation step that some designs will require: examine all interconnections; those that introduce (rather than just transmit) failure conditions need to be re-characterized as components.

3.1 Assumptions and Inputs

The main assumption is that we have a system architectural design that may be preliminary in component or interconnection details but is complete in the following ways:

1. All system modes are identified.
2. All components and their interconnections are identified.
3. A hazard analysis has been performed to identify all component failure conditions that have the capability to propagate outward. The failure conditions are characterized using some kind of error taxonomy [SAE 2015].

In the next section (Section 3.2), we more precisely explain some of these points, leading to the Complexity Formula in Section 3.3.

3.2 Terminology Used in the Formula

The terminology we use in expressing and discussing the formula is as follows:

- A **mode** is an operational phase of a system, during which the system is expected to experience different inputs and behave differently than when in other modes.
  - Examples of modes include takeoff, flying, and landing; or full-strength, partial, and degraded; or weekday and weekend.
  - This definition for mode generally agrees with common use, but the complexity formula only refers to system modes and not component modes. How to adjust the formula to address the situation is discussed under Generalizations, Limitations, and Discussion (Note 3).
- A **propagation point** (P-point) is any interface feature of a component through which a failure condition experienced by that component can emanate out to, and influence the correct functioning of, other components.
  - Examples of P-points include output ports, sockets, bus connectors, and possibly just a region of the component's surface through which that influence might occur.

  - Within a given system mode, a failure condition can potentially emanate out through none, one, or many of that component's P-points. The failure condition is said to be "bound to" this subset of a component's P-points.
  - In common use, a design will identify inward and perhaps bi-directional propagation points as well, but when applying the complexity formula, focus strictly on those propagation points through which failure conditions can propagate out, whether or not they can also be a conduit for the entry of failure conditions.
- **Interconnections** are relations indicating in a given system mode which P-points of which components of a system are connected to which other components. Interconnections convey failure conditions from the component in which they arise through one or more P-points of that component to other components.
  - Note that interconnections are only a relation and are only active during system modes. In some design languages, interconnections can experience failure conditions [SAE 2015] and/or can be active during mode transitions [Feiler 2012]. How to adjust the formula to address these situations is discussed in Section 3.5 (Notes 4 and 5, respectively) and Section 3.6.
  - Some design languages may be more flexible, but we impose this constraint: in each mode, each P-point can have at most one interconnection (in any given mode). We do allow an interconnection to be connected to more than one P-point, which will transmit any failure conditions bound to any of its P-points to the same set of other components. We also allow multiple interconnections from a P-point, but no two of them can be active in the same mode.

3.3 Complexity Formula

Given a system architectural design (including fault model) that specifies:

- m **system modes**: M[i] (i in 1..m), indicating for each:
  - Which system components and interconnections are active
  - Which failure conditions a system component may experience
- n **run-time components**: C[j] (j in 1..n), indicating for each:
  - Its q[j] **P-points**: P[j,k] (k in 1..q[j]), where q[j] = number of C[j]'s P-points)
  - Which failure conditions are bound to which P-points and for which system modes
  - In which modes each component is active
- **interconnections** between P-points and components.
  - Note that for a given system mode, which components a failure condition can initially reach can be determined by tracing the failure condition through any P-points it is bound to (and thus can emanate from) and then on to all components reachable from those P-points, as determined by the interconnections active in that same mode.

This formula also requires definitions of two factors as follows:

- **OutPropagateFailureCount**(i,j,k) =
  number of failure conditions that can propagate out P-point P[j,k] when C[j] is active in mode M[i]; otherwise 0.
- **FanOut**(i,j,k) =
  in mode M[i], the number of components that P[j,k] can transmit a failure condition to, through an active interconnection. (There are either 0 or 1 active interconnections for P[j,k] in mode M[i]; if there are none, than FanOut(i,j,k) = 0.)

The potential size of a safety argument (number of distinct cases that may need to be considered) is then:

```
Sum over all system modes (i in 1..m),
        Sum over all components (j in 1..n),
                Sum over all P-points of C[j] (k in 1..q[j]), of
                        [ OutPropagateFailureCount(i,j,k) * FanOut(i,j,k) ]
```

3.4 Rationale

Arguing that a system is safe based on a run-time characterization of its components and their interconnections potentially requires considering:

- every system mode (M[i]),
- every system component C[j] active in that mode,
- every failure condition that one of those component's might experience that the component cannot be assumed to fully handle (and thus may propagate out), and
- every component that escaping failure condition might propagate to,

then arguing how each component that receives a failure condition addresses it (fully resolves, mitigates, ignores, or transfers it, etc.). Each of these resulting arguments is potentially a distinct case requiring a distinct argument, and thus the formula is structured to consider each of the bullets above and count all the resulting arguments.

3.5 Generalizations, Limitations, and Discussion

During development and piloting of the Complexity Formula, and the subsequent analysis of the results, we made the following observations (referred to as Notes):

1. The **system environment** in its entirety should be one of the system components (one of the C[j]); otherwise, any failure conditions arising in the system environment and propagating into the system will be excluded from consideration. In the examples considered thus far (in our very early testing of the Complexity Formula), what to use as P-points has not been in doubt, and is unlikely to be an issue when the Complexity Formula is applied to subsystems embedded in a larger engineered system. However, derivation of P-points for the system environment could become an issue when applied to a system that directly interacts with people or the atmosphere, for example, unless the design includes a careful and precise definition of the system boundary.

2. The formula is **not symmetric**: it is focused only on **outgoing propagation**. It ignores connectivity into a component and what failure conditions might propagate into that component or be generated by the functioning of the component itself. This limitation is deliberate and intended to **treat the components at the same level of a system design as a "black box,"** in which certain failure conditions can be assumed to be fully handled by the implementers of that component (and thus support compositional verification). In addition, it avoids some forms of double counting and helps to keep the formula simple. As we gain more experience with the formula, we may want to revisit this decision, in which case, the formula can be revised to include both incoming and self-generated failure conditions.
3. A system's **component can have its own modes** nested within a particular system mode—or perhaps independent of the system's modes. In the case of nested component modes, it is best to revise one of the summed items, replacing the contribution of a given component in a given system mode with a sum of the contributions by that component over all its modes that fall within that particular system mode. In the case of a component having modes independent of the system's modes, it may be necessary to consider pairs of modes (or more generally, n-tupes of modes).
4. If any of the interconnections can introduce (i.e., not just pass through) a failure condition, then that **interconnection should be re-characterized as a run-time component** (one of the C[j]) rather than an interconnection; and the rest of the system design topology (components, P-points, and interconnections) should be treated accordingly. (See Appendix B for a recommended way to handle this situation.)
5. Likewise, if a particular interconnection is active only during a mode transition, then for purposes of applying the formula, **treat the mode transition as its own distinct mode** (one of the M[i]).
6. Related to Note 2, the formula explicitly **ignores internal component complexity**. For example, a number of state variables or computations could introduce error. This is one of the issues that exhaustive unit test or verification in DO-178B/C [RTCA 2012] is intended to address. The choice to focus on outgoing propagation implies that this formula applies to the integration of components. Nonetheless, a weight could be applied to each component. The weight could perhaps be the likelihood of error. Alternatively, if a component is itself designed as a system of subcomponents, its complexity could be used as a weight.
7. As long as each of **the system's components are implemented faithful to the analyzed system design**, the formula need not be re-applied. If any of the components of the system design are later implemented in a way that does not conform to the system design (e.g., failure conditions that were originally assumed to be handled by the component in fact "leak out"), then the design and associated fault model should be revised and the formula applied again.
8. It is possible that several of the distinct arguments mentioned under Rationale can in fact be **more efficiently argued together** within a system safety argument, thus reducing the number of truly distinct cases to consider. When this is the case, the summations can be re-factored to sum over equivalence classes.

9. **Use a failure condition (error) taxonomy** (e.g., [SAE 2015]) **appropriate to the domain and design being considered**. A very coarse error taxonomy will result in a smaller number of situations to consider, but larger (and possibly much more convoluted) individual arguments. The error taxonomy used should be both relevant (appropriate to domain and design) and **sufficiently granular to distinguish failure conditions that might differ in impact or resolution**; and should be applied uniformly across the system design. The selection (and any tailoring) of the error taxonomy and its subsequent application should be performed by knowledgeable safety analysts.
10. **Automate** the formula where possible.
11. **As we learn more** from using the formula on other examples, we may want to update the formula to address some of the issues noted (especially, Notes 2 and 6) and the formula may be revised.

4 Procedure for Applying the Complexity Formula

In this section, we present a procedure for applying the complexity formula of Section 3. The procedure is organized as a sequence of steps that a safety analyst or a reviewer of an architecture might manually employ on a small or medium-sized system. The formula has been applied independently by the authors of this report to the design of a small system and the same result was obtained 3 out of the 4 times it was applied (see Section 7). Therefore, rather than applying this procedure as-is on a larger system, we recommend the formula be automated for application to larger system designs (Section 5).

Starting Point. The key input to applying the formula is the design and fault model for a system. The design must include (or be used to derive) a run-time representation of the system as a set of interconnecting components because it is to that view that the complexity formula will be applied. The fault model should be an elaboration of the design using an (appropriate tailoring of an) error taxonomy, such as [SAE 2015], to help ensure consistent identification of failure conditions across the system design.

If the design or fault model is incomplete. When not all the information needed to apply the formula or this procedure is available, averages based either on historical data or on what is known of the rest of the design topology, or estimate or ranges obtained using a consensus-based decision-making approach (e.g., the Delphi Method) can be used in place of missing parameters (such as the number of failure conditions that can propagate out, or the fan-out).

Here are the steps:

Step 1. Determine how many system modes there are and assign them labels.

Let m represent the total number of system modes and label them M[1] to M[m].

Modes generally have explanatory names such as those given in the definition of “mode” given in Section 3.

We will generally index modes with the variable “i,” which can vary from 1 to m, and we will generally just refer to modes by their label and index.

Step 2. Determine how many components there are and assign them labels.

Include the system environment as one component. The environment “component” takes as input any data that the system outputs, and provides as output any data that the system needs to input, or receives, from the environment. The system boundary with the environment should be reviewed to be sure it is complete; and assumptions documented.

Interconnections that can introduce their own failure conditions should be re-characterized as components (Appendix B).

Let n represent the total number of components and label them C[1] to C[n].

Like modes generally, components may have explanatory names, but we will not refer to them in this procedure. Instead, we will index components with the variable “j,” which can vary from 1 to n, and we will refer to components by their label and index.

Note which components are active in each mode.

Step 3. For each component, determine how many outward propagation points (P-points) it has, and assign them labels.

For each component C[j], let q[j] represent the total number of its P-points.

We will index the P-points of a component C[j] with j and the variable k. In the case of component C[j], this means k can vary from 1 to q[j]; and thus we will label the P-points P[j,1] to P[j,q[j]].

Step 4. Having identified the modes, components, and P-points; create for Mode[i] where i from 1 to m, a drawing depicting a run-time view of the system in mode M[i].

For each mode M[i], create Drawing[i]:

a. Determine how many components are active in mode M[i]. “Active” means they convert inputs to outputs, or more generally, generate any outputs. This includes outputting a data value to memory/storage.

   The total number of active components will be labeled n[i] for mode M[i]. The components that are active may or may not be the same for every mode. Generally, startup and degraded modes will have a different number of components that are active than will normal operational modes.

   For clarity of the steps that follow, we will index the n[i] components that are active in mode M[i] with i and with variable r. In the case of mode M[i], this means r can vary from 1 to n[i]; and thus we will label the active components in mode M[i]: D[i,1] to D[i,n[i]].

Thus, every component in the run-time view of the system should now have at least two labels, and perhaps many more. First, it is, uniquely, one of the C[j], for j in 1..n; which we call its corresponding main index. Second, for each mode M[i] in which it is active, it also has an additional label D[i,r], for r in 1..n[i].

b. Put into Drawing[i] each component that is active in that mode, and each of its P-points.

   The environment is typically active in every mode. Do not forget to include any of the interconnections re-characterized as components in step 2.

   Note that the P-points of an active component D[i,r] can still be labelled P[j,k] for k in 1..q[j], and where j is the corresponding main index.

   (Recall that P-points belong to a component and are thus active whenever the component is considered active. However, just because a component is active does not mean that each of its P-points necessarily has an associated interconnection.)

c. Include in Drawing[i] each interconnection that is active in that mode, displaying the P-points in its domain and the components in its range.

   For each P-point P[j,k] of a component D[i,r] active in mode M[i], there are either 0 or 1 active interconnections to other components.

   If there is an active interconnection, label it A[i,r,k] ("A" for arc). Then indicate that P[j,k] is in its domain (there may be other P-points in its domain that can be indicated in the diagram at the same time that you consider P[j,k]) and indicate the set of components in its range. (For example, it may be possible to simply visually show A[i,r,k] connecting P[j,k] to the components in the range of A[i,r,k]; and likewise for other P-points in its domain.)

   Whenever the P[j,k] has no active interconnections, indicate that this is the case, such as with ∅ representing the empty set. This will help clarify whether or not any failure conditions bound to P[j,k] can actually propagate out to affect other components. (Note: check for an error in the design or fault model if failure conditions are bound to a P-point but there are no mechanisms, that is, interconnections, for those failure conditions to actually be transmitted anywhere.)

Step 5. Given Drawing[i] that now depicts the run-time view of the system in mode M[i], we can at last apply the Complexity Formula (Section 3.3).

By our terminology (Section 3) and definitions above, if P[j,k] has no active interconnections that connect to it, then FanOut(i,j,k) = 0. But if there are active interconnections, then the range of A[i,r,k] is the set of components that a failure condition leaving P[j,k] can reach, and thus:

size (range (A[i,r,k,s])) = FanOut(i,j,k).

Also by our terminology:

size (set of failure conditions bound to P-point P[j,k]) =
OutPropagateFailureCount(i,j,k)

And now the formula can be applied.

Applying the procedure

The procedure above was largely devised by one of the authors as preparation to applying the formula to the stepper motor example (Appendix A). The same author then maintained a log while stepping through the procedure, which appears below. It has been edited to make it more succinct. [1]

I see only one mode. I expected to see a "Startup mode" but none is mentioned. So the only mode appears to be the operational mode. In the notation used to express the complexity formula, we'd say there's only M[1].

I see 3 regular components (PCS, ACT, and Motor), one interconnection that needs to be re-characterized as a component (Bus 2), and one component that is the environment. Hence 5 components:

- C[1] is Environment
- C[2] is PCS
- C[3] is Bus 2
- C[4] is ACT
- C[5] is Motor.

Each has only one P-point therefore k=1. Also, as there is only one mode, and all components and interconnections are active in that one mode, we'll dispense with the D[i,r] labels and just use the C[j] labels for j in 1..5 to represent the 5 components. Each component has a unique P-point and it is associated with exactly one active interconnect.

- For C[1], there is 1 P-point and 3 failure conditions hence Sum over all P-points of C[1] is 3.
- For C[2], there is 1 P-point and 5 failure conditions hence Sum over all P-points of C[2] is 5.
- For C[3], there is 1 P-point and 3 failure conditions hence Sum over all P-points of C[3] is 3
- For C[4], there is 1 P-point and 3 failure conditions hence Sum over all P-points of C[4] is 3
- For C[5], there is 1 P-point and 3 failure conditions hence Sum over all P-points of C[5] is 3

Hence the sum over all 5 components, in the only mode, is 17.

1 The log shown here has been modified to simulate the log that might have been kept if the fault model associated with the original stepper motor design had been correct. See Footnote 3 in Appendix A for a more detailed explanation.

5 Automating the Complexity Formula

As mentioned in the previous section, before applying the complexity formula to a large system design and associated fault model, we recommend developing a program to automate the complexity formula.

The procedure given in Section 4 clearly can be automated if the design and fault model to which the formula is to be applied to are specified in a language having a formal and consistently-used syntax (e.g., for AADL for the design language and EMV2 [SAE 2015] for the error taxonomy that is the basis for the fault model).

Automating the Complexity Formula would involve some effort. Design languages (e.g., AADL [Feiler 2012]) can be fairly complex provide multiple ways to represent the same design construct through various abstraction mechanisms, thus requiring that any program able to correctly interpret designs specified in that language selecting from a large variety of syntactic constructions and binding rules.

Even on a relative small (if complex) example, the stepper motor (Appendix A), a certain amount of tedious work was necessary, and one member of the author team made a small error and obtained an answer different from the other three team members. Automation would have prevented this error.

There is no intent in the current project to automate the complexity formula but we do think that doing so is important prior to applying it to many larger examples.

6 Using the Complexity Formula

6.1 Guiding the Development of the Architecture

How might a system or software architect or fault model developer benefit from the Complexity Formula?

The overall complexity of an emerging system design and fault model could be continually tracked and perhaps displayed during design and fault model development. While not all the parameters of the formula would be available until late in architectural design and fault modeling, estimates or proxies for the missing parameters can be provided from multiple sources including historical data, subject matter experts, and statistics from the current design's network topology (e.g., average fan-out), as mentioned at the beginning of Section 4. Such a capability would almost certainly require automation (Section 5).

Because complexity is a predictor of effort, and in some circumstances, human error, the Complexity Formula also has a potential role in system trades. In system trades, the formula could be applied to evaluate alternative architectural designs (with missing parameters estimated) and provide a rough indicator of how much effort will be required (whether as part of the system trade or if selected, later by

a stakeholder reviewer) to review the alternative for safety. The premise is that more complex alternatives engender higher risk of misunderstanding and drawing wrong inferences.

Not just reviewers (Section 6.2) but also developers, testers, suppliers, and system maintainers all need to review the system design documentation in the future; and for some of the same reasons, a higher value returned by the formula might imply higher development, integration, test, and operations and maintenance effort.

6.2 Estimating the Effort Required to Review a Safety Claim

How might the reviewer of a system architecture benefit from the Complexity Formula?

For a reviewer who has maintained data on how long it takes to consider each distinct case (a failure condition propagating out of one run-time design component and reaching a second component), the estimate provided by the formula can be multiplied by the average "consideration time" to obtain a rough estimate of how much effort a systematic review of a safety argument will require.

Ideally, such consideration-time averages and ranges might serve as benchmarks, and be specific to the domain, product line, developer and/or architect.

Before an architectural reviewer initiates their review, the benchmark would serve as a starting point, together with the result returned from the Complexity Formula, to estimate the amount of effort required to perform a review. To this estimate should be added an estimate of how much time will be required to do any required research, pose questions, and interpret answers from the system developer or manufacturer. (Alternatively, if the reviewer already has been maintaining an effort log of previous safety claim reviews, they can use that data to derive an initial estimate of consideration time.) As the review progresses, the reviewer can then replace the benchmark with their own consideration-time average reflecting effort expended on the review of that particular system design and fault model, and update their effort estimate. If the reviewer continues to track their time allocation to review, research, pose questions, and interpret answers, their consideration-time average can be periodically updated and effort remaining likewise updated. At the end, the reviewer can contribute their time and effort logs to help in maintaining the consideration-time benchmark(s).

6.3 Minimum Knowledge Required to Apply the Complexity Formula

If the Complexity Formula is automated, then of course not much knowledge is required of the complexity formula itself, though some knowledge of its parameters would be valuable if it is to be repeatedly used during architecture development (Section 6.1) in order to anticipate the effects of design and fault model changes.

If the Complexity Formula is to be applied manually, the procedure described in Section 5 should first be tailored to reflect the specifics of the design and fault modeling language used (e.g., can interconnections be active in some modes but not others). Thus, the applier of the formula needs to understand that design language. Of course, the applier also needs to understand the formula, whose expression is not so complex but Section 5 shows that there are many relatively tedious steps that need to be applied

correctly. It is also important to have access to someone who understands the rationale for the system design and fault model if the system design does not come already include such information.

Of course, if a reviewer who will be applying the formula manually (Section 6.2, above), then that reviewer should be an expert in the domain(s) touched on by the system design and especially knowledgeable of potential failure conditions.

7 Notes on an Example Application

We considered two designs for the Stepper Motor System. In this section, we will summarize our results from applying the Complexity Formula to the two designs. While a detailed understanding of the designs is not necessary to understanding this section, we recommend scanning Appendix A, which describes the original design for the stepper motor, as well as the inter-rater reliability experiment that was conducted. (The new design is described in [Feiler 2015] and is not included in this report.)

The original design of the stepper motor had software system SM_PCS send incremental commands to SM_ACT based on an assumed position for the stepper motor. (There is no back channel from the stepper motor informing SM_PCS of its position or confirmation that it has received or executed a command sequence.) This design was the focus of our inter-rater reliability experiment.

The revised design had software system SM_PCS send an absolute position to SM_ACT. The revised design has much simpler software. However, using this design requires a change to the stepper motor hardware.

In the original design, the motor is given a number of steps to move and a direction. Its logic counts down that value every time it makes a step and stops when it gets to zero (without regard to its current position). In the revised design, the logic must compare the current position to the desired position after each step. When they match, it stops moving.

What do we get in terms of reduced software complexity for this relatively simple hardware logic change (and required changes to other components to adjust to the hardware logic change)? Here is a list of evidence that needs to be collected for the original design that does not need to be collected for the revised design.

- Evidence that the stepper motor homes to the fully closed position at startup.
- Evidence that the SM_PCS accurately knows where the stepper motor really is at the start of each frame.
- Evidence that the stepper motor will always have time to move S steps within a frame.
- Evidence that calculated clock accuracy matches actual clock accuracy
- Evidence that the execution of SM_PCS happens no more frequently than F.

Obtaining some of the above evidence is quite difficult.

On the other hand, there is no evidence that must be collected for the revised design that does not have a substantially similar piece of evidence required to be collected for the original design.

Applying the Complexity Formula to both designs results in 17 for the original design and 16 for the revised design. That is, reviewing a safety claim for either system design could require reviewing 16 or 17 distinct cases where an error propagates from a first component to a second.

8 Conclusion

We have a formula for estimating potential size of a safety case for a system based on the degree of modes, fan-out, and number of propagating failure conditions that can be identified in a run-time design view of the system.

The formula has been applied to a small, but non-trivial example and tested. Three out of 4 team members were able to apply the formula correctly; the fourth had one count off by one, but found the error easily after a review.

We conclude that although the formula can be applied manually, automation would improve use of the formula.

We considered how the formula would be used, both as a guide to developing the system architecture, and in estimating the effort required to review a claim of system safety.

If the formula should also detect the impact of small changes in design that can nevertheless have a disproportionate impact on effort, then the formula should be expanded to weigh, in some manner, the severity of the error propagations, rather than implicitly assuming "all error propagations are created equal." We hope to investigate this issue in the future.

Appendix A Inter-Rater Experiment

According to [Wikipedia 2016], "**inter-rater reliability**, **inter-rater agreement**, or concordance is the degree of agreement among **raters**. It gives a score of how much homogeneity, or consensus, there is in the ratings given by judges." In this appendix, we describe the inter-rater reliability experiment that was conducted to evaluate the degree to which the complexity formula could be correctly applied by raters to an unfamiliar example. The motivation was to assess whether the formula could be applied by others to a system design and fault model and whether the same result would be obtained by the different raters.

The raters in this case were two other members of the team who were already exposed to an earlier version of the complexity formula that lacked some of the factors appearing in the final version. To make the exercise a bit more challenging, the explanation of the formula itself was limited to a single printed page. Prior to the actual experiment, the two members of the team preparing the experiment applied an earlier version of the instructions to the same example that they were preparing for the rest

of the team and obtained the same answer (16)[2]. This gave them confidence that the other two team members who had not been experienced the recent evolution of the formula nor had encountered the example before might get the same answer.

Components of the Experiment

- Cover letter of instructions, including reporting results
- Complexity formula
- Example that the formula was to be applied to

The two members of the team preparing the experiment created a summary version of the design and fault model for a stepper motor. This took some effort, as the design and fault model were scattered over multiple pages of a technical report and the intent was to make the example self-contained and compact.

Here are the instructions. Where detail not directly relevant to the conduct of the experiment has been omitted, the phrase "[snip]" appears. Also, the names have been changed to "Team Member N."

> Team Members 1 and 2,
>
> [snip] Please read both attachments and apply the formula to the example. Please do so without speaking to each other until you have both completed the exercise and submitted to Team Members 3 and 4 your results. If you have the time, you might also critique the one-page formula description. (If you want to additionally critique the stepper motor example, you can do that too, but the focus should really be on the one-pager).
>
> Team Members 3 and 4 worked hard to get the stepper motor example description down to two pages. [snip] The handout created largely reflects Team Member 4's understanding of that system and its design but is also informed by a careful analysis of the AADL model.
>
> The complexity formula has been revised in two critical respects since we last discussed it with you:
> (1) We distinguish the ways in which a failure condition might propagate outward from a component by introducing the concept of **propagation points** [snip].
> (2) We completely abstracted the formula from (made it agnostic to) the particular architectural design language being used to characterize the system's interconnections and error flows. [snip].
>
> [Sent by] Team Members 3 and 4

2 The original version of the stepper motor design (used in the inter-rater reliability experiment) had one failure condition missing from the design, and thus the consensus number shown above was 16. Subsequently, the missing failure condition was identified, making the correct output of the formula 17. We are confident that if we ran the experiment again, a similarly consistent result would be obtained across the different raters.

Here is the description of the Complexity Formula that was provided with the instructions:

Estimating the Potential Size of a System Safety Argument (aka, "Complexity Formula"), V017

Given an architectural design for a system expressed in terms of:

- m **system modes**: M[1]..M[m]
  - The design must identify which components and connections (C&C) are **active** in which modes.
  - If a particular connection is active only during a mode transition, then for purposes of applying the formula, treat the mode transition as its own distinct mode (one of the M[i]).
- n **run-time components**: C[i] (i in 1..n); their **propagation points** P[i,k]; and **interconnections**
  - Treat the system environment in its entirety as its own component (one of the C[i]).
  - Designate the propagation points (PP) of C[i] as: P[i,1]..P[i,q] (q depends on i).
  - If any of the interconnections can introduce (i.e., not just pass through) a failure condition, then treat that connection as a run-time component (one of the C[i]) and update the topology (PPs and interconnections) accordingly.
- For each C[i], P[i,k], and M[j], **define**:
  - **OutPropagateFailureCount**(i,j,k) = number of failure conditions that can propagate out through PP P[i,k] when C[i] is in mode M[j].
  - **FanOut**(i,j,k) = number of components that PP P[i,k] connects (directly) to in mode M[j].

The potential size of a **safety argument** (number of distinct cases that may need to be considered) is:

```
Sum over all system modes (indexed by j ranging over 1..m),
    Sum over all components (indexed by i ranging over 1..n),
        Sum over all PPs of C[i] (indexed by i and k),
            OutPropagateFailureCount(i,j,k) * FanOut(i,j,k)
```

Rationale in a nutshell: Arguing that a system is safe based on a run-time characterization of its components and their interconnections potentially requires considering:

- every system mode (M[j])
- each system component C[i] active in that mode
- the failure conditions that C[i] can experience in that mode M[j] that can propagate outward
- through which propagation points P[i,k] of C[i] that failure condition might propagate through
- which interconnections (active in that mode M[j]) connect to that (those) P[i,k] and carry that failure condition to which other component (or the system environment)
- and whether those receiving components then address that failure condition in some way that is satisfactory

Notice that each such consideration (i.e., the failure conditions and number of ways they can propagate outward) can potentially require a distinct line of argument, and thus the number of such considerations provides an estimate of the potential size of (number of situations that may require specific consideration in) a safety argument.

Limitations/Possible Improvements to the Formula:

(1) Several such situations (of failure propagation) might efficiently be considered together, reducing the argument's size. When this is the case, the summations can be re-factored to sum over equivalence classes. Be cautious about doing this as subtle differences in propagation points, interconnections, etc. may require distinct argumentation.

(2) Use a failure condition (error) taxonomy appropriate to the domain and design being considered. A very coarse error taxonomy will result in a smaller number of situations to consider, but larger individual arguments. The error taxonomy used should be both relevant (appropriate to domain and design) and sufficiently granular to distinguish failure conditions that might differ in impact or resolution; and should be applied uniformly across the system design. The selection of the error taxonomy and its application should be performed by knowledgeable safety analysts.

(3) Automate the formula where possible.

Notes not part of main text:

(1) Assumption: PPs can only be active when the component they are part of is active. More precisely, a failure condition within C[i] cannot propagate through a particular PP of C[i] unless C[i] is active; and it cannot propagate over a particular interconnection unless that interconnection is active. An implication of this is that care needs to be taken in identifying and specifying PPs and interconnections so that the characterization of the system accurately reflects how failure conditions can propagate.

(2) A more complete version of the formula (for use with models that are sparsely inter-connected) includes as a term placed inside the second but outside the third summation: **Failure-Count**(i,j), which equals the total number of failure conditions component C[i] can experience in mode M[j].

Here is the description of the Stepper Motor design and fault model that were provided with the instructions:

Description of the operation of the stepper motor system

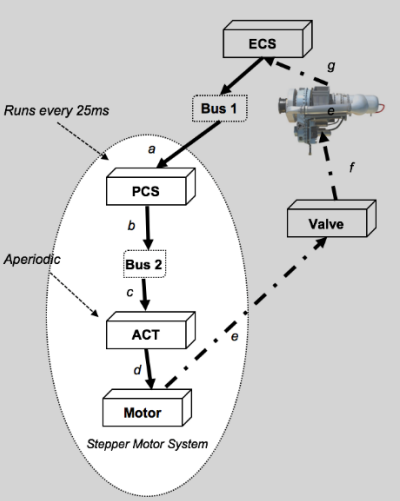


Figure 1: SMS and its environment (software shown as white boxes)

The Stepper Motor System (SMS) consists of a Position Control System (PCS), an actuator (ACT), and a stepper motor (Motor). The SMS receives instructions from the Engine Control System (ECS) over a data Bus (Bus 1) to tell it how to set the fuel valve that meters fuel to the engine.

- At 0%, the intent is that no fuel flows to the engine.
- At 100%, the intent is that the maximum fuel flows to the engine.

The ECS receives no direct feedback from the SMS; rather it infers that the correct amount of fuel is reaching the engine by sensing the speed of the engine.

At power up, the Motor is initialized to fully closed (0). The Motor can subsequently be set to any of N positions where N is much greater than 100.

The PCS (which runs as a periodic task every 25ms) takes the most recent percentage open requested by the ECS (via Bus 1) and converts it to a proportional number between 0 and N. This is the desired position of the Motor.

The ACT takes the number of steps to move and the direction to move from the PCS via direct memory access (represented in the figure as Bus 2) at any time and causes the Motor to so move until either the Motor has moved the requested number of steps, or a new number of steps and direction to move is received from the PCS. The Motor is physically capable of moving no more than S steps during the 25ms frame rate of the PCS.

Since the Motor is only capable of moving S steps during a single PCS frame, the PCS actually issues a sequence of commands to the ACT – one each frame, telling it to move S or fewer steps

in the appropriate direction until it has commanded enough steps to reach the desired position[3]. When the ACT has moved the Motor the requested number of steps, it goes idle until provided with a new command from the PCS. The PCS checks for a new command every 25ms frame. If there is a new position requested, it begins to issue commands to the ACT reflecting that. If there is no new position requested, it continues to issue commands to the ACT to move the Motor to the position previously requested. If there are no more commands to issue, then it does nothing.

Possible failure conditions

The items below correspond to the labeled arrows in the previous figure:

a) Propagating out from Bus 1:
no service, data corruption, data loss

b) Propagating out from PCS:
no command issued, wrong command issued, late command issued, early command issued, and data replication error[4]

c) Propagating out from Bus 2:
no service, data corruption, data loss

d) Propagating out from ACT:
wrong step count, wrong direction, incomplete execution of steps

e) Propagating out from Motor:
motor failure, motor missed step commands, motor slow to respond

The inter-rater reliability experiment used the original design, without the data replication error included for the PCS. Informally summarized, the experiment's actual results were that three of the four team members obtained 16 cases and one team member obtained 17 cases. Review of the latter revealed a spurious entry in a spreadsheet, which he then had interpreted as an additional case, thus all 4 team members agreed the total should have been 16 cases. We are assuming that, had we included the fifth error for the PCS, all four raters reviewing the original design would have determined a complexity score one higher.

Appendix B Re-characterizing selected interconnections as components

In Note 4 of the Section 3.5, we noted that if any of the interconnections in the system design can introduce a failure condition apart from what can propagates into it (in any system mode whatsoever),

---

3 This requires that the PCS keep accurate track of the stepper motor's location at the beginning of each frame. Subsequent to the original distribution of this report, an additional PCS failure condition was identified only applicable to the original design: data replication error (failure to keep accurate track of the stepper motor position). When this failure condition is included when applying the complexity formula, the result is 17, instead of 16.

4 Note that this last error was not included in the rater test described before.

then that interconnection should be treated as a run-time component. In this section, we describe how the system design topology should be adjusted following such a replacement. There are multiple ways to accomplish this; we will provide one.

The only interconnections selected for re-characterization are those that filter the data they receive, maintain some state, or which are physical in nature; and thus might cause either logical or physical errors to anything transmitted over them, or they might generate their own spurious signals or data. Example failure conditions introduced by such an interconnection include data not transmitted, data transmitted early or late, data corrupted between source and sink.

Other interconnections do not need to be re-characterized and thus remain interconnections. We can think of them as "logical interconnections" whose only function is to transmit whatever appears on any of its input termini to all output termini.

The system topology should be revised, for the purpose of applying the complexity formula, following the re-characterization of an interconnection as a component:

- In common usage, we would normally think of an interconnection as a relation (one relation for each mode) among components, or more precisely, among selected "interaction points" of one or more components of a system. However, as the only interaction points considered in the complexity formula are outward-oriented (P-points), we can refine the way we regard an interconnection:

  An interconnection is a mapping from a set of P-points (the domain) to a set of components (the range) in which any failure condition experienced at a P-point in its domain is transmitted to all components in its range.

  (The next assumption is not critical to the presentation of the procedure, but we normally think of interconnections as active or not active; but when active, the same P-points to components relation is defined. If in a given mode, a different set of components can be reached, this should be represented by a different interconnection. Again in any mode, for each P-point there can be at most one interconnection active that it can use.)

- Because re-characterizing an interconnection effectively replaces it with a component, we need to identify the new component's P-points and failure conditions and any interconnections incoming or outgoing (i.e., attached to one of its P-points).
- For each mode in which the interconnection was active in the original design, let FCPI represent the set of all failure conditions that could propagate into that interconnection (this is simply the union, for the specified mode, of all failure conditions bound to any of the P-points in its domain). Let FCIG represent the set of failure conditions that the interconnection could itself generate under the original design.
- Create a single P-point for the new component and bind the union of FCPI and FCIG to it. The resulting set is the set of all failure conditions that can propagate out from the new component through its unique P-point. Notice this is the same set of failure conditions as could propagate out

of the original interconnection for that specified mode. (We want the propagation behavior of the new component to mirror that of the original interconnection.)

- Outward bound, create a single logical interconnection between the single P-point mentioned above to every component in the range of the original interconnection. This new interconnection is active in precisely those modes in which the original interaction was active.
- Inward bound, create a single logical interconnection between every P-point in the domain of the original interconnection to the new component. Again, this new interconnection is active in precisely those modes in which the original interaction was active.

Appendix C Answer Key: Solution for Example in Appendix A

This solution is corrected to add a fifth failure mode for PCS.

To answer, follow the procedure for applying the complexity formula in Section 4.

Step 1: Determine how many modes.

Nothing in the write-up indicates that there are more than one mode: no separate startup, or degraded, or partial (see the definition of "mode" in Section 3.2)

Therefore #Modes = 1.

Step 2: Determine how many components there are and assign them labels.

From the next line, the Environment is one component, "Env." (component #1)

From Figure 1 (in Appendix A), other components are

- PCS (#2)
- Bus 2 (which is a component because the bus between PCS and ACT has ways to fail) (#3)
- ACT (#4)
- Motor (#5)

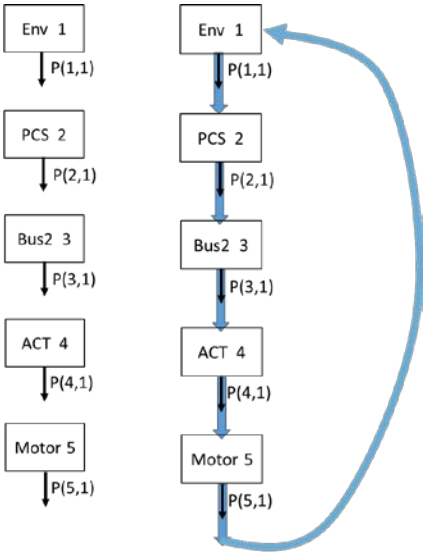
There are five components.

Step 3: Outward propagation points:

From Figure 1:

- Env has 1 (only to PCS), call it EnvOut = P(1, 1)
- PCS has 1 (only to Bus 2), call it PCSout = P(2, 1)
- Bus2 has 1 (only to ACT), call it Bus2out = P(3, 1)
- ACT has 1 (only to Motor), call it ACTout = P(4, 1)
- Motor has 1 (only to Env), call it MotorOut = P(5, 1)

Step 4: Create 1 drawing (1 mode)



a. How many components are active? (all 5) so n(1) (i.e. n for Mode 1) = 5.
   r will thus vary from 1 to 5. Components are labeled C(1, r) so their drawings are labeled D(1, r)
b. Put into the one drawing, each component that is active, and each of its P-points.
c. Include in drawing 1 each interconnection that is active in the mode, displaying the p-points in its domain and the components in its range.

   In this case (Mode 1 = only mode; 5 components, each with only 1 P-point), the interconnections are just continuations of the arrows down to the next component.

According to Step 4 in Section 4, "We can at last apply the Complexity formula."

To do this we need to know what are the number of out-propagations possible for each interconnection, and the fanout.

Since there is only one interconnection per propagation point, fanout is 1 in all cases. So now all we need is to determine how many ways can a failure propagate out of each component (each propagation point, since there is one per unit).

Per Appendix A, the OutPropagationFailureCounts for each component are:

- Component 1 (Bus 1, which is part of Env), 3 failure conditions possible
- Component 2 (PCS), 5 failure conditions.
- Component 3 (Bus2), 3 failure conditions.
- Component 4 (ACT), 3 failure conditions
- Component 5 (Motor), 3 failure conditions.

(The steps and answers above are all discussed in short form on page 12).

So to do the summations we apply the formula in the box on page 7:

Sum over 1 mode

Sum over 5 components

Sum over 1 p-point each

(OutPropagationFailureCount * 1)

Or, more briefly

Sum over 5 compontents

(OutPropagationFailureCount)

In other words,

Comp. 1 OutPropagationFailureCount +
Comp. 2 OutPropagationFailureCount +
Comp. 3 OutPropagationFailureCount +
Comp. 4 OutPropagationFailureCount +
Comp. 5 OutPropagationFailureCount

Or 3 + 5 + 3 + 3 + 3 or 17.

The complexity formula yields a complexity of 17 possibly propagating failures.

References

URLs are valid as of the publication date of this document.

[Feiler 2012]
Feiler, P. & Gluch, D., *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Part of the SEI Series in Software Engineering series. Addison-Wesley Professional, 2012 (ISBN-10: 0-321-88894-4).

[Feiler 2015]
Peter H. Feiler, Charles B. Weinstock, John B. Goodenough, Julien Delange, Ari Z. Klein, Neil Ernst. "Improving Quality Using Architecture Fault Analysis with Confidence Arguments." CMU/SEI-2015-TR-006. March 2015.

[Goodenough 2013]
Goodenough, John, Weinstock, Charles, & Klein, Ari. "Eliminative Induction: A Basis for Arguing System Confidence." *International Conference on Software Engineering (ICSE 2013)*. San Francisco, CA, May 2013. IEEE/ACM, 2013.

[Konrad 2015]
Michael Konrad and Sarah Sheard, "FAA Research Project: System Complexity Effects on Aircraft Safety. Literature Review Task 3.2: Literature Search to Define Complexity for Avionics Systems." CMU/SEI-2015-SR-006, v.2, May 2015.

[Nichols 2015]
Nichols, William R. and Sarah Sheard. "FAA Research Project: System Complexity Effects on Aircraft Safety. Task 3.3: Candidate Complexity Metrics. CMU/SEI-2015-SR-012, May 2015.

[RTCA 2012]
RTCA, Inc. *DO-178C Software Considerations in Airborne Systems and Equipment Certification,* January 2012.

[SAE 2015]
SAE International. SAE AS-5506/1A:2015, SAE Architecture Analysis and Design Language *(AADL) Annex Volume 1: Annex A: ARINC653 Annex, Annex C: Code Generation Annex, Annex E: Error Model Annex*. 2015.

[Sheard 2015]
Sarah Sheard, Chuck Weinstock, Michael Konrad, and Donald Firesmith, "FAA Research Project: System Complexity Effects on Aircraft Safety. Task 3.4: Identify the Impact of Complexity on Safety. CMU/SEI-2015-SR-022, July 2015.

[Wikipedia 2016]
Definition of Inter-rater reliability. https://en.wikipedia.org/wiki/Inter-rater_reliability.

Acknowledgments

The SEI team thanks the FAA team for many useful discussions in understanding how safety is evaluated today, and especially, in alphabetical order: Mike DeWalt, Barbara Lingberg, and Srini Mandalapu. We also thank our SEI colleagues, including Julien Delange, Peter Feiler, and Dave Zubrow for their suggestions during the development of this report.

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-0004272