Software Engineering Institute
Carnegie Mellon University

FAA RESEARCH PROJECT ON SYSTEM COMPLEXITY EFFECTS ON AIRCRAFT SAFETY: TESTING THE IDENTIFIED METRICS

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

May 2016

Executive Summary

The purpose of this report is to describe a test of the complexity algorithm that we developed and previously described in the report for Task 3.5: *Estimating Complexity of a Safety Argument* [Konrad 2016]. The algorithm did not measure every possible kind of complexity; rather it measured the complexity resulting from errors propagating from component to component; thus we call it *error propagation complexity*. The example used in this test was a Wheel Brake System that had an available model in the Architecture Analysis & Design Language (AADL), a much more involved example than the stepper used in the previous report. Its error propagation complexity turned out to be about twice that of the Stepper Motor System.

The method used to assess error propagation complexity for the Wheel Brake System was to obtain the architecture model of the entire Wheel Brake System, simplify the interconnections, then (as in Konrad 2016) count the ways that errors could propagate from one element to another. The initial model is shown in Figure 1 of this report, and the simplified model is Figure 3. The Wheel Brake System was chosen because it had a fairly complete model, including a description in a formal architectural language (AADL) that noted system modes, identified components and their interconnections, and had a reasonably complete error model (resulting from a hazard analysis).

The architecture was simplified to focus on aspects important to application of the error propagation complexity formula, namely, modes (this system had only one), components, propagation points, failure conditions, and fan-out. As before, the error propagation complexity formula essentially estimates the size of the safety case: assuming an average analysis time for the follow-through to determine whether a failure can propagate in an unsafe manner, the estimate of total time for safety case analysis can be created by multiplying this average time per failure propagation by the number of ways a failure can propagate, which is estimated by our formula for error propagation complexity.

From the diagram shown in Figure 3, a complexity value can be reliably computed, as validated in an inter-rater test. The answer key in Section 5 shows how this is done. It consists of seven steps plus a matrix organizing their answers.

In conclusion, this report shows that the formula for error propagation complexity can be applied consistently to multiple well-defined architectures and results in reasonable answers.

1 Introduction

This special report describes the results of Task 3.6, 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:

> *3.6 Test the Identified Metrics*
>
> *Test the identified metrics on a jointly-agreed-to representative avionics system to prove the applicability of the proposed approach. Analyze 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.*
>
> *Deliverables:*
>
> *White paper reporting the relevance of selected metrics and demonstrating the applicability of our methods to manage complexity.*

Some work originally conceived of for this task was also reported in [Konrad 2016] when the complexity of the stepper motor example was tested to proof the algorithm.

2 Selecting the Example

2.1 Wheel Brake System

We initially applied the error propagation complexity[1] formula to a Stepper Motor System [Konrad 2016]. To further validate the formula and understand how it might be used in practice, we tried it on a larger example. Section 3.1 of the previous report lists the assumptions and inputs that guided the selection of our example:

> *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 from a component**. The failure conditions are characterized using an error taxonomy.* [Konrad 2016, emphasis added]

The team identified two examples that met most of these criteria (components, interconnections, and failure conditions are identified—there were no modes specified): a Wheel Brake System for an aerospace system [Feiler 2014, SAE 2011] and a Speed Regulation System for a car control system

1 Now called error propagation complexity; this quantity was just called complexity in the previous report.

[Delange 2015]. Although the documentation of the architectural design for the speed regulation system was more complete in some ways, we determined that this completeness would not affect our analysis, and thus we chose the Wheel Brake System as being more relevant to the avionics domain. We call the hazard analysis the "error model" in this work.

The Contiguous Aircraft/System Development Process Example describes the Wheel Break System as follows:

> *This AIR [Aerospace Information Report] describes, in detail, a contiguous example of the aircraft and systems development for a fictitious aircraft design. In order to present a clear picture, an aircraft function was broken down into a single system. A function was chosen which had sufficient complexity to allow use of all the methodologies, yet was simple enough to present a clear picture of the flow through the process…the principles used at the braking system level can be applied at the higher aircraft level*...[SAE 2011]

The SEI's AADL Wiki contains several architectural designs for the Wheel Brake System. We selected one of the simpler ones [Feiler 2014].[2] The example provides two architectures that are identical functionally but differ in the platforms they are deployed to: Integrated Modular Avionics (IMA) and Federated. The IMA version uses a single processor with four partitions (or virtual processors) to execute the software. The Federated version uses several processors interconnected through a bus. We selected the IMA version, though the analysis for the Federated version would be similar.[3] Figure 1 depicts the IMA version of the Wheel Brake System.

2 Available from the AADL Wiki at https://wiki.sei.cmu.edu/aadl/index.php/Simple_version_of_the_ARP4761/AIR6110_example

3 The Federated example is discussed briefly in Appendix B, Alternate Case: Federated System.

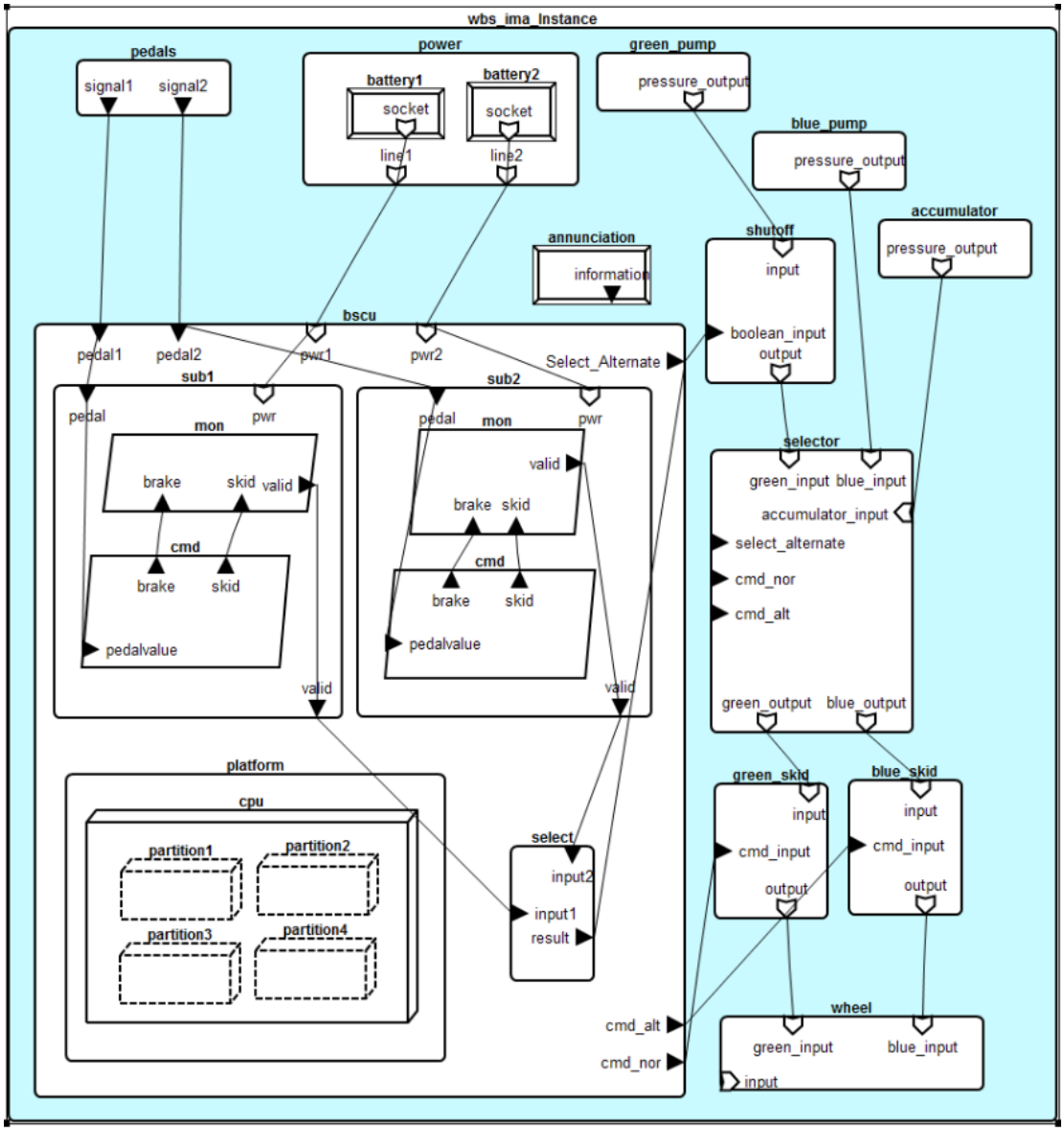

Figure 1: IMA Version of the Wheel Brake System

2.2 Finding a More Complete Specification of the Example

Figure 1 cannot be used as is for applying the formula for these reasons:

1. The underlying fault model is not depicted; in particular, the figure by itself does not identify how many failure conditions can propagate out of which components to affect other components.
2. The figure does not show how the Brake System Control Unit (BSCU) software is deployed onto the platform. The deployment determines how hardware and operating system-level failure conditions propagate to the software.
3. The figure appears to be missing a few interconnections.

To obtain a more complete model of the IMA version of the Wheel Brake System, we retrieved the AADL model (see Figure 2 for a short excerpt) from the SEI GitHub repository where it is maintained (identified at [Feiler 2014]) and analyzed that model to determine how best to manually apply the error propagation complexity formula.

```
system implementation wbs.detailed extends wbs.generic
subcomponents
    selector   : refined to system valves::selector_detailed{Classifier_Substitution_Rule => Type_Extension;};
    shutoff    : system valves::boolean_shutoff;
    green_skid : system valves::cmd_shutoff;
    blue_skid  : system valves::cmd_shutoff;
    wheel      : refined to system wheel::wheel_two_inputs.i {Classifier_Substitution_Rule => Type_Extension;};
connections
    --  We connect the blue pump directly to the selector valve.
    --  For the green pump, we connect it to the shutoff valve that
    --  is later connected to the selector valve after.
    blue_to_selector    : bus access blue_pump.pressure_output  <-> selector.blue_input;
    green_to_shutoff    : bus access green_pump.pressure_output <-> shutoff.input;
    shutoff_to_selector : bus access shutoff.output <-> selector.green_input;
    bscu_to_shutoff     : port bscu.Select_Alternate -> shutoff.boolean_input;

    bscu_to_selector    : port bscu.Select_Alternate -> selector.select_alternate;

    --  Connect the command from the BSCU to the skid valves
    bscu_to_gskid       : port bscu.cmd_nor -> green_skid.cmd_input;
    bscu_to_bskid       : port bscu.cmd_alt -> blue_skid.cmd_input;

    --  Connect the output pressure from the selector
    --  to the anti-skid components
    selector_to_gskid   : bus access selector.green_output <-> green_skid.input;
    selector_to_bskid   : bus access selector.blue_output  <-> blue_skid.input;

    --  Connect the anti-skid system to the wheel. In this version,
    --  we see the wheel as having two paths for getting the pressure.
    --  We see that as a physical model, not a logical one.
    bskid_to_wheel      : bus access blue_skid.output <-> wheel.blue_input;
    gskid_to_wheel      : bus access green_skid.output <-> wheel.green_input;
end wbs.detailed;
```

Figure 2: AADL Model of the Wheel Brake System

3 Preparing to Apply the Error Propagation Complexity Formula

3.1 Why Some Level of Preparation Is Needed

Automating the error propagation complexity formula for direct application to an AADL model [Konrad 2016, Section 5] is a nontrivial undertaking, so some up-front analysis is required to identify those model features addressed by (and that need to be input to) the error propagation complexity formula.

Also, the model contains a lot of specification detail whose relevance to the error propagation complexity formula can be difficult to determine unless the person applying the formula is fluent in AADL (including its Error Model Annex).[4] Such detail can be confusing and distracting to the person applying the formula manually and can obscure the essential features of the design that contribute to its

4 The design and fault model specification for the Wheel Brake System encompasses about a dozen separate AADL packages.

complexity. Of course, such detail is important if the goal is to have a model with sufficient detail that it can be analyzed for satisfaction of requirements, but not all such detail is needed when applying the error propagation complexity formula.

This level of detail presented a challenge: *How can we express the features of the Wheel Brake System design relevant to applying the formula in a succinct way so that the formula can be directly and straightforwardly applied by someone not familiar with a specific architecture description language?* Though the approach we outline in this section specifically applies to AADL models, it should generalize to models using other architecture description languages. Having a more succinct representation of the design also enables others to review how the formula was applied without having to be familiar with the original language that expressed the design.

3.2 Some Observations That Help Simplify the Example

Having asked ourselves what a more succinct, non-AADL depiction might look like, we reviewed the model in some detail. We observed several characteristics of this model:

1. Most run-time components have only a single outbound propagation point.
2. Most components have only a single failure condition that can propagate outward from that component.
3. None of the interconnections can experience failure conditions.

Although these observations might have been violated if we had found a higher-fidelity, more fully specified model of the Wheel Brake System, we proceeded with the available model. We anticipate that during certification of a real system, a more extensive model will be developed that would correctly specify propagation points and error conditions.

3.3 Steps for Simplifying the Model

Figure 3 shows our simplified model. It was created by adding detail missing in Figure 1 that is essential to the error propagation complexity formula while eliminating the extraneous detail shown in that figure. This allows us to determine the complexity directly with only a minimum of off-the-figure information. Details of the simplification are described in the subsections that follow.

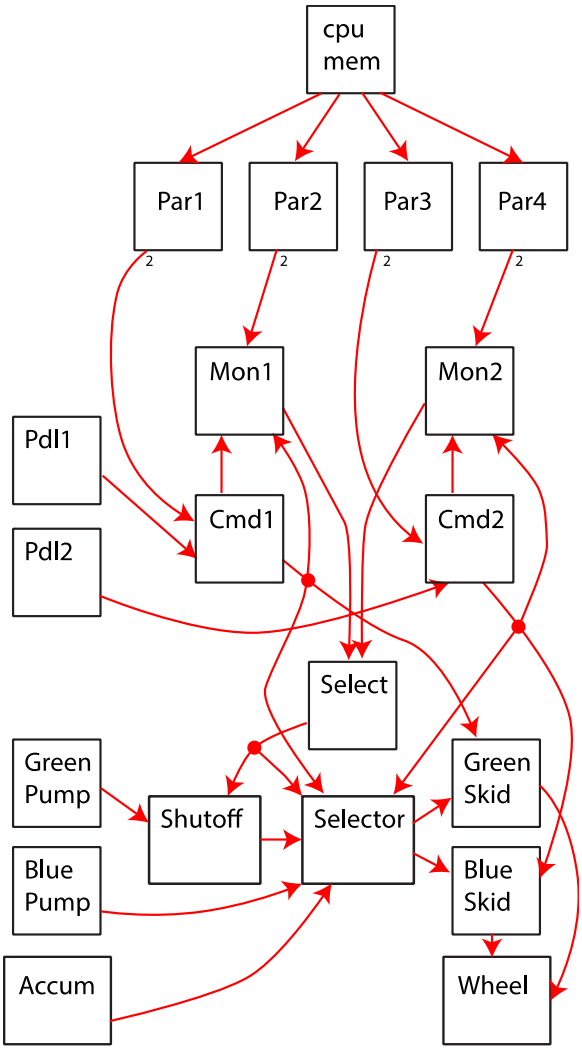

Figure 3: IMA Version of the Wheel Brake System Clarified for Application of the Error Propagation Complexity Formula

Step 1. Characterize the Deployment of the Software to the Platform

We redrew the model in Figure 1 using the AADL model for the IMA version as a guide, being careful to explicitly characterize the deployment of the software to the platform.

The computational infrastructure is shown as five components in Figure 3:

- *cpu mem*: the CPU and its memory and partition control software (also shown as *cpu* in Figure 1)
- *Par1..Par4*: the four partitions that the CPU memory is divided into. The two command processes (*Cmd1* and *Cmd2*) and two monitor processes (*Mon1* and *Mon2*) are assigned to different partitions; also shown as *partition1...partition4* in Figure 1. However, Figure 1 does not show what software components are deployed to which partition. These relationships are shown by connections in Figure 3.

Examining the error model embedded within the AADL model, we see that the *cpu mem* can experience a hardware failure, which can propagate to one or more of these four partitions. The partitions can forward this failure condition to the software processes that are bound to them (*Cmd1* and *Cmd2*; *Mon1* and *Mon2*). In addition, the partitions can experience a software error and propagate this failure condition to the software processes bound to them.

In Figure 3, the error model is shown as four propagation points on *cpu mem*, through which a single failure condition (hardware failure) can propagate out (one or more of them), and a single propagation point on each partition, through which two failure conditions (hardware failure and software failure) can propagate out to the software processes bound to them.

Step 2. Remove All Higher-Level System Representations That Group Run-Time Components Together

To reduce clutter in the figure, we removed all five of the higher-level system representations that group several related run-time components together. The result is shown in Figure 3. Further explanation is given below:

- These higher-level system representations impart no run-time information and thus slightly inflate the "score" returned by applying the error propagation complexity formula without significantly affecting the effort expended in reasoning about system safety. We chose to eliminate all five of the higher-level system representations (platform, pedals, Wheel Brake System, BSCU system, and the two BSCU subsystems), while retaining the lower-level run-time components that they contained.
- For example, the platform system serves to demarcate the computational infrastructure from the BSCU software that is deployed to it. Removing this demarcation decreases the number of hops that failure conditions propagating from the CPU and partitions must cross to affect the software running on them and thus clarifies what is actually happening. The computational infrastructure is not so complex that there is benefit to keeping the platform as an enclosing system, and therefore it was eliminated.
- While it does not affect applying the error propagation complexity formula, the correct way to read Figure 3 is that *cpu mem* has four propagation points with a single failure condition associated with each, versus a single propagation point with four connections.

Step 3. Remove Orphaned Inputs

After some debate, we removed the battery *pwr* inputs from the figure. They were inputs to the BSCU system that were orphaned once we removed that system (and in any case were not carried further within that system). Thus there is no contribution to a safety-claim assurance case, assuming the error model and architecture are complete and correct. (A more complete model of the Wheel Brake System would more fully specify the distribution of power—after all, power can contribute additional failure conditions that we should care about. We chose not to second-guess what the architect might have designed for power distribution to the Wheel Brake System. (For an example, see [Haskel 2016].)

Step 4. Introduce Fan-Outs to Replace Multiple Connections from the Same Propagation Point

We introduced fan-out indicators (red dots serving as arc connectors in Figure 3) wherever, according to the AADL model, there are multiple connections from the same propagation point. In addition to removing some of the arc congestion, the introduction of fan-out indicators enables more direct use of the formula, which requires knowing how many failure conditions can propagate out through which propagation points on a component and the fan-out of each propagation point. There are three fan-out indicators in Figure 3:

- *Cmd1* (and *Cmd2* likewise) has two propagation points: one propagation point positioned at the top of *Cmd1* and one positioned at the right (bottom) edge of *Cmd1* (and *Cmd2*). The one positioned at the top connects directly to *Mon1* and corresponds to the brake outbound data port in the AADL model. The one positioned at the right (bottom) edge fans out directly toward three components via a fan-out indicator:
  a. the *Mon1* process to enable skid monitoring—yes, there are two connections between *Cmd1* and *Mon1*
  b. the *Green* (and *Blue*) *Skid* valve for skid control
  c. the *Selector* valve. Thus, *Cmd1* has two propagation points, one with a fan-out of 1, with no fan-out indicator, and one with a fan-out of 3, depicted by a fan-out indicator.
- *Cmd2* likewise has two propagation points, one with a fan-out of 1 with, of course, no fan-out indicator and one with a fan-out of 3, depicted by fan-out indicator.
- Finally, there is the fan-out from the Select device to two components: the *Shutoff* valve and *Selector* valve. The fan-out is 2.

Step 5. Eliminate Propagation Point Symbols and Labels

In Figure 1, data ports are shown by a solid triangle; bus access points representing power connections and hydraulic pressure valve connections are shown as a clear chevron; and all are labeled. Such labeling is helpful in an architectural description; however, to apply the formula to a run-time depiction (architectural view) of the system, it is necessary to determine only the number of failure conditions associated with each propagation point and how many places such associated failure conditions can propagate to. The former can be visually depicted by having separate arcs emerge from a component, one for each propagation point (we will visually show only outbound propagation points), and by placing a small positive integer next to the outbound arcs to indicate the number of failure conditions associated with each propagation point.[5] The latter can be visually depicted by simply connecting the arc to the appropriate components, using a fan-out indicator (Step 4) when the propagation point propagates to more than one component.

We can therefore eliminate from a visual run-time depiction (architectural view) of the system all data port (and access point) symbols and labels, replacing them with visually distinct propagation points on

5 We adopt the convention of not showing the number of failure conditions when the number is 1.

the component's surface and integer indicating the number of failure conditions associated with each propagation point. This method allows us to simplify the depiction of the design and does not require any additional off-figure information to apply the formula.

Below, we identify every component in the Wheel Brake System that has more than one propagation point and what, if any, further simplifications were taken (only the depiction of pedals was further simplified):

- *cpu mem*: See Step 1.
- *Pedals*: We replaced the *pedals* component in Figure 1, which had two outbound data ports, *signal1* and *signal2* (and would have been another exception to Observation 1), with two separate run-time components, *Pdl1* and *Pdl2*, each having a single propagation point. This simplification has no impact on the complexity score and better portrays what happens at run-time.
- *Command* processes: In the AADL model, each *Command* process has two outbound data ports: *brake* and *skid*. We described how we addressed these components and their propagation points above and in Step 4.
- *Selector* valve: In the AADL model, the *Selector* valve has two ports from which an arc leaves and connects to either the *Green Skid* valve or the *Blue Skid* valve. There is no ambiguity in the visual depiction.

Step 6. Eliminate Failure Condition Names

When applying the error propagation complexity formula, there is no need to have a detailed accounting of the error model. It is not even necessary to provide any off-the-figure information as long as we maintain the convention described in Step 5: distinguish propagation points that have more than one failure condition associated with them by labeling the associated arc with a count of the number of failure conditions associated with that propagation point. Propagation points that have zero associated failure conditions have no reason to be shown in the figure, if our only intent is to apply the error propagation complexity formula (unless we expect that number to change, in which case the propagation point or its arc should be labeled with a 0). Thus, the convention is sufficient for correct application of the error propagation complexity formula.

Step 7. Depict the Run-Time System Visually (for the General Case)

In the general case (i.e., for examples other than the Wheel Brake System), there may be connections that can themselves experience failure conditions. How to modify the model used to address such connections in preparation for applying the error propagation complexity formula is described in Appendix B of our previous report [Konrad 2016]. In the case of the Wheel Brake System, there are no such connections.

In the general case, there may be multiple system modes. Both the Stepper Motor System and Wheel Brake System have only a single system mode specified (implicitly). In the general case, there can be one visual depiction of the run-time view per system mode. In this case, we apply the formula to each mode separately and then sum the individual mode-specific scores to provide an overall complexity score for the system design.

Step 8. Examine the Error Model for the Components of the Wheel Brake System

As described in previous sections, there is no need to consider at a detailed level the error model for the system prior to applying the formula; only some counts are needed. Nevertheless, to give some idea of the complexity inherent in the error model for the Wheel Brake System as summarized in Figure 3, we present a detailed accounting of all components appearing in Figure 3 (or omitted from Figure 3 per Sections 3.3.2 and 3.3.3) and their status relative to Observation 2 of Section 3.2:

- Buses: Power and hydraulic pressure buses (declared as *power*, *power.generic*; *pwm*, *pwm.generic*; and *pressure*, *pressure.i* in the AADL model) introduce no additional failure conditions (Section 3.2).
- Power batteries (*pwr1*, *pwr2*) comply with Observation 2 of Section 3.2 (a propagation point has a single failure condition); they can only propagate *NoPower*. A detailed look at a battery's error model reveals two error events: *battery depleted* and the catastrophic *battery explodes*.[6] Regardless of which event happens, with respect to the portion of the aircraft being modeled, the battery state transitions from Operational to Failed, and *NoPower* propagates out. These inputs were eliminated as described in Step 3.
- Pedal signals (*Pdl1, Pdl2*) comply with Observation 2; they propagate only *NoService*.
- *Command* processes comply with Observation 2 because when any failure condition propagates in, the *Command* process goes into a Failed state and outputs (only) *NoValue* on both ports (*brake* and *skid*).
- *Monitor* processes comply with Observation 2; they propagate only *NoValue* through the one outbound port.
- *BSCU* system was eliminated as a component per Step 2.
- *BSCU* subsystems: As mentioned in Step 2, components that are actually just containers—high-level system or subsystem representations decomposed into internal components—really have no functional role in a safety argument and thus can be removed before applying the formula.
- *Select*: The *select_alternate* system, which is how the *Select* component is declared in the AADL model, complies with Observation 2 of Section 3.2, propagating out only *NoValue* through its one output port.
- *Platform* was eliminated as a component by Step 2. A review of its declaration indicates that it introduces no new failure conditions to the two mentioned in Partitions (one of which is originally propagated out of the CPU).
- *Partitions* does not comply with Observation 2 of Section 3.2. Each of the four partition components (*Par1*..*Par4* in Figure 3) can propagate two failure conditions outward. In the AADL model, these are labeled *HardwareFailure* and *SoftwareFailure*. To apply the formula correctly, we do not need to know anything more about these failure conditions or the propagation points other than that there are two of them. This fact is depicted in Figure 3 by the small label 2 on the arcs exiting the partition components.

---

6 *Battery Open Circuit* was not included in the error model; in any case, the result would be the same "no power."

- *CPU* complies with Observation 2. The CPU (*cpu.ima* in the AADL model) has only one failure condition, *HardwareFailure*, and according to the associated component error behavior model, it propagates that failure condition through its bindings when it occurs. Thus, the CPU complies.
- The pumps *Green Pump, Blue Pump*, and *Accumulator* and the valves *Green Skid* and *Blue Skid*) all comply with Observation 2. Pumps have a single failure condition, *NoService*, that they potentially propagate out the single propagation point when an internal hydraulic failure occurs. Some valves simply propagate out the failure condition (*NoService*) when it propagates into the valve; for example, a type of valve named *boolean_shutoff* does this. Another type of valve named *cmd_shutoff* in the AADL model likewise permits *NoService* failure conditions to flow through, but when its additional input data port *cmd_input* receives a *NoValue* failure propagation, it propagates outward a *NoService* failure condition. Thus, in this case, too, there is compliance with Observation 2.
- *Selector* system (not to be confused with the *Select* component that takes inputs from both BSCU subsystems), called *selector_detailed* in the AADL model, is the component with the highest total fan-in and fan-out in the entire Wheel Brake System. There are
  - three input data ports: two from the command processes and one from the *Select* component
  - five hydraulic access points: from the *Green Pump, Blue Pump*, and *Accumulator* as inputs and to the *Green Skid* and *Blue Skid* as outputs
  - two types of failure conditions that can propagate in: *NoValue* (from the three input data ports) and *NoService* (from the pumps)
  - only one failure condition that can propagate out (*NoService*)

  Thus, the *Selector* also complies with Observation 2 of Section 3.2.
- *Wheel*: All variants of the wheel system declared in the AADL model propagate out 0 failure conditions. Essentially, the *Wheel* marks the external system interface with the environment.
- Annunciation: The *Annunciator* displays ("announces") recent status of the multiple systems on which safe operation of the aircraft depends. The *Annunciator* can experience a *LossAnnunciation* error event that propagates *NoService* out through its sole output data port, and thus it complies with Observation 2 of Section 3.2. However, because the *Annunciator* is not directly connected to anything in the figure, it is not shown. (There is 0 fan-out from the *Annunciator*'s only output data port; thus there's no reason to include it in the error propagation complexity formula because its contribution to the total score would be 0.)
- Wheel Brake System: The Wheel Brake System is composed of all the above systems and was eliminated as a separate component in Step 2. A review of its declaration indicates that it introduces no new failure conditions. It does, however, have a more complex error behavior model with multiple states, reflecting the state of the wheel brake and whether or not its state is being correctly annunciated to the pilot, but such a consideration is tangential to our purpose and does not enter directly into the application of the error propagation complexity formula.
- Environment: We have no explicit single environment component in the Wheel Brake System example, unlike the Stepper Motor System. We varied somewhat from the guidance in our previous report [Konrad 2016] by choosing to represent the interfaces of the Wheel Brake System

with the environment explicitly as distinct components (*Pld1*, *Pdl2*, the three pumps, and *Wheel*). This has no effect on the score returned by the error propagation complexity formula.

3.4 The Clarified Run-Time Model for the Wheel Brake System

After pursuing the approach outlined in Section 3.3, we have a simplified (clarified), single-page visual depiction of the system, shown in Figure 3. It is sufficient for applying the formula without additional off-the-figure information, as long as the person applying the formula is familiar with conventions such as arc labeling and fan-out indicators (Steps 5 and 6). As described in Step 7, the approach can be generalized to architectures specified in other architecture description languages, multiple modes, and connections that can fail.

4 Applying the Error Propagation Complexity Formula to the Wheel Brake System

In the previous section, we discussed how we developed Figure 3, a simplified version of the Wheel Brake System architecture that is sufficiently detailed for application of the error propagation complexity formula.

Recall that the potential size of a safety argument (number of distinct cases that may need to be considered) is

> 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
> [$OutPropagateFailureCount(i,j,k) \cdot FanOut(i,j,k)$]

We now have enough information to calculate the complexity of the Wheel Brake System.

Steps to calculate the value of the error propagation complexity formula from the simplified model

1. Count the number of system modes. Consider each mode separately in the following steps.
2. Determine how many components there are (*C*).
3. Determine how many propagation points there are for each component (*PP*).
4. Determine how many failure conditions can propagate from each propagation point (*FC*).
5. Determine the fan-out from each propagation point (*FO*).
6. For each propagation point, calculate the value of $FC \cdot FO$.
7. Add the values obtained from this calculation for each propagation point.

The resulting number is the complexity value for the mode under consideration. If there are multiple modes, add the values obtained for each mode to obtain the system complexity value.

In the next section, we show our calculation for the Wheel Brake System, but interested readers are invited to make their own calculation before proceeding.

5 Answer Key

In this section, we determine the value of the error propagation complexity formula for the Wheel Brake System example shown in Figure 3 using the steps outlined in the previous section.

1. In this example, there is only a single mode to consider.
2. Counting the number of components in the Figure, $C = 20$.
3. Each of the 20 components has a single propagation point, except for *cpu/mem* (4), *Selector* (2), *Wheel* (0), and *Cmd1* and *Cmd2* (2 each).
4. Each of the propagation points can propagate a single failure condition, except for the partitions (*Par1* to *Par4*), which each can propagate 2.
5. Each of the propagation points has a fan-out of 1 except as follows:
    a. One of the two propagation points in each of *Cmd1* and *Cmd2* has a fan-out of 3.
    b. The propagation point in *Select* has a fan-out of 2.

With this information, we can easily calculate the value of $FC \cdot FO$ for each propagation point.

Summing the values from the previous step gives an error propagation complexity value of 34. Thus the estimated number of potential safety issues due to error propagation is 34. All of these cases will have to be examined separately in order to determine that this system is safe. This number is not terribly useful in itself, but the comparison to other systems is useful. So it is fair to say that the Wheel Brake System is about twice as complex (from a safety standpoint) as the Stepper Motor system was, since that had an error propagation complexity of 16.

The following table summarizes the calculations (*P* = propagation; *FC* = failure conditions, *FO* = fan out).

Component Type	Number of Components	Number of P Points per Component	#*FC*	Fan-Out	#Components * #P points * #*FC* * *FO*
cpu mem	1	4	1	1	4
Par1..Par4	4	1	2	1	8
Mon1..Mon2	2	1	1	1	2
Cmd1..Cmd2	2	1	1	1	2
		1	1	3	6
Pdl1..Pdl2	2	1	1	1	2
Select	1	1	1	2	2
Green Pump, *Blue Pump*, *Accum*	3	1	1	1	3
Shutoff	1	1	1	1	1
Selector	1	2	1	1	2
Green Skid, *Blue Skid*	2	1	1	1	2
Wheel	1	0	n/a	n/a	0

Components: 20 Error Propagation Complexity: 34

6 Conclusion

We have developed a formula for estimating the potential size of a safety case for a system, based on the number 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 two nontrivial examples and tested. In the first example, three out of four team members applied the formula correctly; the fourth made an error that was found and easily corrected after a review. In the second example, once the model was simplified, all team members came up with the same value.

Based on our experiences, we conclude that although the formula can be applied manually, automation would improve use of the formula and would be possible once a suitable diagram has been developed manually.

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.

We also explored why the formula might not have detected a difference in complexity between an original design and a new design of the same system. Primarily, this is because the formula was developed to use system design or fault model parameters that are available early, and thus drive effort as designs grow more complex. It is our view that the formula succeeds as it identifies complexity independent of detailed-design.

So that the formula may be used to detect the impact of small changes in design that can nevertheless have a disproportionate impact on effort, the formula should be expanded to include another parameter, one that reflects the number of inter-component dependencies (constraints) that arise from system design choices. In the original stepper motor example, an architect made an assumption in the original design that one component "knew" the position of another component, which results in a cascade of possible failure conditions that need to be checked. This assumption was removed from the new, simpler design that we analyzed had an error propagation complexity of 16.

Appendix A Abbreviations

Abbreviation	Definition
AADL	Architecture Analysis & Design Language
accum	accumulator
AIR	Aerospace Information Report
alt	alternative (i.e., backup)
ARP	Aerospace Recommended Procedure
bscu / BSCU	Brake System Control Unit
cmd	command
cpu	central processing unit
FAA	Federal Aviation Administration
FC	failure condition
FO	fan out
ima / IMA	Integrated Modular Avionics
mem	memory
mon	monitor
nor	normal (i.e., primary)
Par	partition
Pdl	pedal
PP	propagation point
pwr	power
SAE	Society of Automotive Engineers
skid	anti-skid[7] operation
sub	Wheel Brake Subsystem
wbs	Wheel Brake System

7 Anti-skid operation is achieved by regulating hydraulic pressure that is applied to the brakes in such a way that wheel rotation is maintained [Haskel 2016].

Appendix B Alternate Case: Federated System

The Federated Version of the Wheel Brake System is identical to the IMA Version except in two respects: (1) the Federated version uses a double CPU configuration rather than a single CPU configuration for its implementation infrastructure; and (2) the software is mapped (or deployed) to the computational infrastructure differently. Because of the similarity between the two versions, the simplification described in Section 3 leads to almost the same figure for the two versions and almost the same error propagation complexity table. Compare Figure 4 to Figure 3, and the table below Figure 4 with the table in Section 5. The only differences in the two figures are in the first two rows of components that compose the computational infrastructure and their connections to the software underneath. Due to the operating system being modeled directly as part of the CPU (rather than in separate partition components as in the IMA version), the Federated Version has a smaller number of components and lesser error propagation complexity.

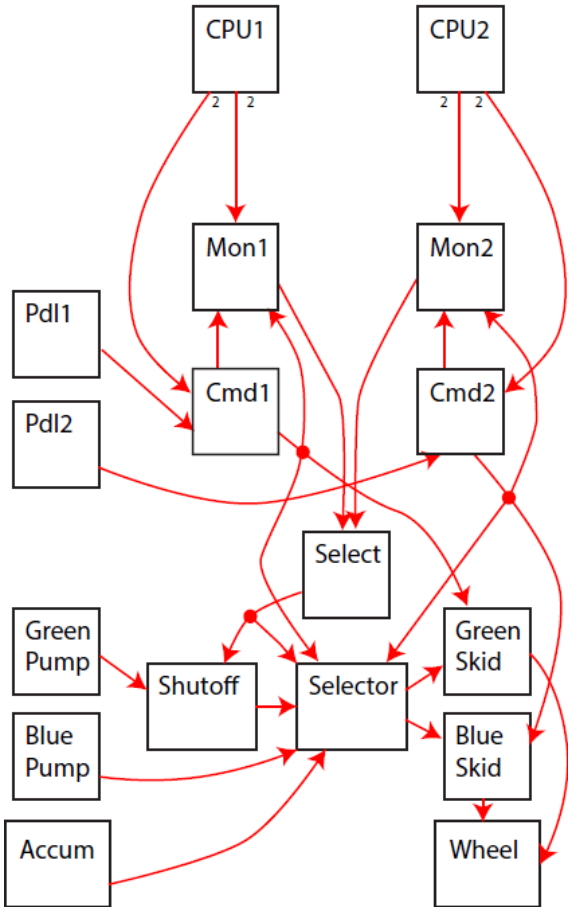

Figure 4: Federated Version of the Wheel Brake System Clarified for Application of the Error Propagation Complexity Formula

The following table summarizes the calculations (*P* = propagation, FC = failure conditions, *FO* = fan out).

Component Type	Number of Components	Number of P Points per Component	*#FC*	Fan-Out	#Components * #P points * *#FC * FO*
CPU1..CPU2	2	2	2	1	8
Mon1..Mon2	2	1	1	1	2
Cmd1..Cmd2	2	1	1	1	2
		1	1	3	6
Pdl1..Pdl2	2	1	1	1	2
Select	1	1	1	2	2
Green Pump, *Blue Pump*, *Accum*	3	1	1	1	3
Shutoff	1	1	1	1	1
Selector	1	2	1	1	2
Green Skid, *Blue Skid*	2	1	1	1	2
Wheel	1	0	n/a	n/a	0

Components: 17 Error Propagation Complexity: 30

References

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

[Delange 2015]
Delange, Julien. “Speed Regulation.” AADL Wiki. Software Engineering Institute, Carnegie Mellon University. 2015. https://wiki.sei.cmu.edu/aadl/index.php/SpeedRegulation

[Felier 2014]
Peter Feiler. “Wheel Brake System.” AADL Wiki. Software Engineering Institute, Carnegie Mellon University. 2014. https://wiki.sei.cmu.edu/aadl/index.php/ARP4761_-_Wheel_Brake_System_%28WBS%29_Example

[Haskel 2016]
Eddie Haskel. “G450 Systems.” Code 7700. 2016. http://code7700.com/g450_landing_gear_wheels_and_brakes.html

[Konrad 2016]
Konrad, Michael; Sheard, Sarah; Weinstock, Chuck; & Nichols, William R. “FAA Research Project: System Complexity Effects on Aircraft Safety. Task 3.5: Estimating Complexity of a Safety Argument.” CMU/SEI-2016-SR-007. Software Engineering Institute, Carnegie Mellon University. February 2016.

[SAE 2011]
SAE International. “AIR 6110, Contiguous Aircraft/System Development Process Example.” 2011. http://standards.sae.org/air6110/

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 Administratioin 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-0004273