Harmony aMBSE Deskbook Version 1.00

Agile Model-Based Systems Engineering Best Practices with IBM Rhapsody

Bruce Powel Douglass, Ph.D.
Chief Evangelist
Global Technology Ambassador
IBM Internet of Things

bruce.douglass@us.ibm.com

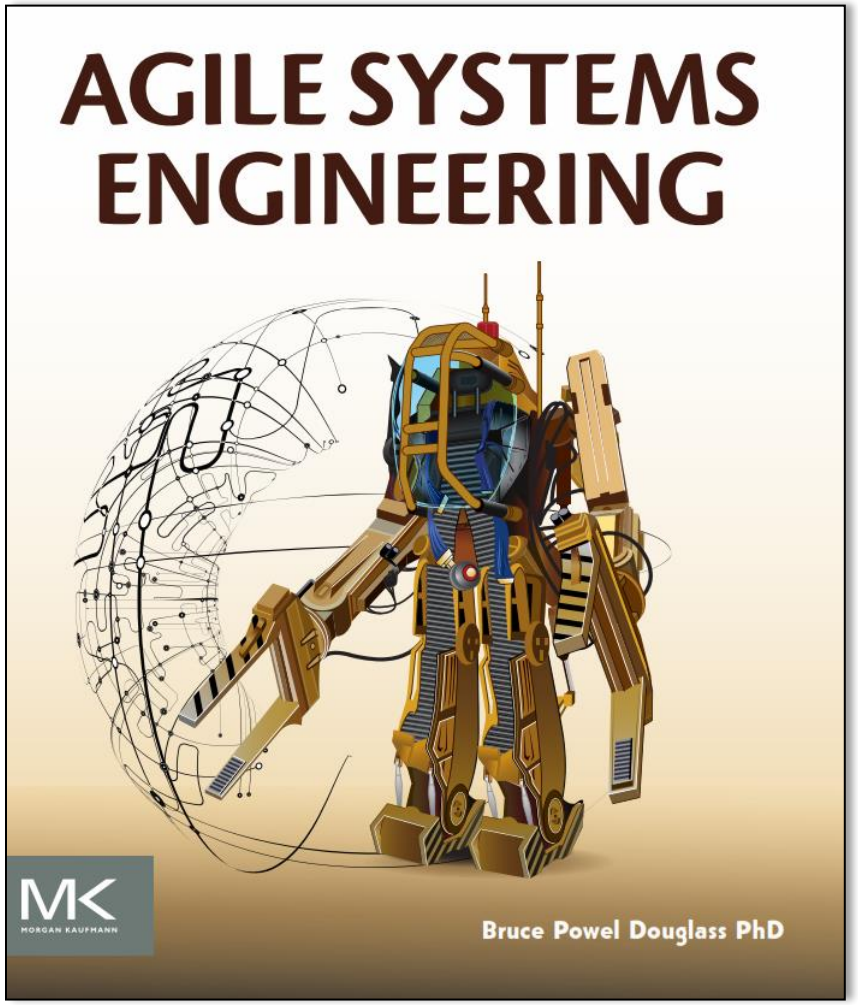
Black Edition:
Rhapsody Only

This is the latest version of the Harmony aMBSE Deskbook, released September, 2017.

This Deskbook is written for the systems engineer. This Deskbook assumes the reader is familiar with

- Systems engineering concepts
- The SysML language
- The IBM Rhapsody UML/SysML Modeling Tool

IBM Corporation
Software Group
Route 100
Somers, NY, 10589
USA



1 Foreword

This Deskbook provides guidance, "best practices", for using model-based systems engineering in an agile way. This deskbok is based on the method outlined in my book *Agile Systems Engineering* (Elsevier Press, 2016), seen at the left, which is, in turn based on the previous Harmony for Systems Engineering and Harmony for Embedded Software work. Readers wanting more detailed exposition are referred there for more detail. Although based heavily on that book, this Deskbook differs in a number of important ways.

- The Deskbook is considerably lighter in depth and breadth, compared to a full book.
- The Deskbook does not introduce the SysML, Rhapsody tool, nor, in any detail, agile methods as they apply to systems engineering in general. It does, however, briefly introduce the work flows and work products of the Harmony aMBSE process.
- The Deskbook is intended primarily as means to get system engineers quickly up to speed using the approach without a great deal of theoretical and historical backstory.
- The Deskbook is meant to introduce the best practices in the context of a process (the Harmony aMBSE process), a SysML tool (IBM Rhapsody), and a particular example system.
- The Deskbook provides mentoring on the use of the Rhapsopdy tool, and especially the use of the Harmony SE Toolkit, written by Andy Lapping.
- Finally, the Deskbook is free. ☺

Although this Deskbook is written by me, I wish to acknowledge the significant contributions of Graham Bleakley, Ph.D., and Andy Lapping, both of IBM.

I have tried hard to remove all errors in this Deskbook. Despite that effort, I have no doubt that some remain. If you discover an error, please report it to me via email at Bruce.Douglass@us.ibm.com.

This Deskbook was created using the Rhapsody Developer Edition version 8.2.1.

2 Table of Contents

3 Introduction

3.1 Why this Deskbook?

To enable effective systems engineering, a number of things are necessary, most notably 1) language, 2) process, and 3) tooling. Ultimately, the purpose of this Deskbook is to show how to best unify these aspects together into a holistic, efficient, and effective systems engineering practice. Let's talk about these three key aspects of a systems engineering practice.

A Systems Engineering Language

First, a language is needed to capture the semantic elements and their relations. Natural language has its place; it is wonderfully expressive and easy for non-technical people to understand, at least in general terms, what is being said. It is a great way to capture poetry or to discuss the nuances of philsophical arguments. Nevertheless, it is problematic for systems engineering. It is ambiguous, and the same word often not only means different things to different people, it often means several different things to the same person. Natural language is imprecise because even if a word has a precise meaning, it is likely to have subtle aspects. In general, natural language is not computable, or at least not in the same way as mathematics or temporal logic are. Natural language sacrifices precision for universality. This is a good tradeoff if you want to write a haiku, but a bad one if you want to describe the laws of physics.

SysML, on the other hand is a more precise language with a metamodel specification (http://www.omg.org/spec/SysML). It includes a number of representational views for functionality (use case and requirement diagrams), structure (internal block and block definition diagrams), behavior (activity and state diagrams), interaction (sequence diagrams) and relations (various table and matrices). These views adhere to an underlying semantic model so that their meaning is precise enough to create **computable models**.

Computable models are important because they allow the verification of the information the hold. An important subset of computable models are **executable models** – models that can be executed or simulated to verify they correctly capture semantic content. Since the primary outcome of systems engineering activities is specification, computable models permit the engineer to verify the correctness of the information within the model as well as to validate, with the customer, that the system under development will meet their needs. This can be done with virtually all systems engineering work products, from requirements specifications to architecture trade studies, architectural specifications, interface specifications, and other work products handed off to downstream engineers.

A Process for Planning and Enacting Engineering Work

A process is a procedure that specifies **what** you want to do, **when** you want to do it, **what** you need to consume and create, **who** needs to be involved, and **how** to go about it.

In this context, the **Harmony Agile Model-Based Systems Engineering** (**aMBSE**)[2] process defines all those things and provides guidance on how to proceed. aMBSE is *agile* because it incorporates some key agile approaches to optimize both correctness of the work and to minimize the effort required. aMBSE is *model-based* because it relies on SysML and computable modeling to identify, represent, and verify the system properties of concern. aMBSE is for *systems engineering* in that it focuses on the specific needs of systems engineers. The Harmony aMBSE process will be discussed in more detail in the next chapter.

A Tooling and Automation Environment

From one perspective, tools are nothing particularly special. They merely automate things that you would normally do via more manual procedures. However, good tools do more than just save time; they also improve quality, and in the best case, empower the engineer to perform activities that, while desirable, were unachievable before.

In this Deskbook, the tool of concern is IBM Rhapsody, a highly capable UML/SysML modeling environment. Logically, Rhapsody consists of a number of interconnected capabilities that collectively provide a powerful conceptual place from which to develop systems.

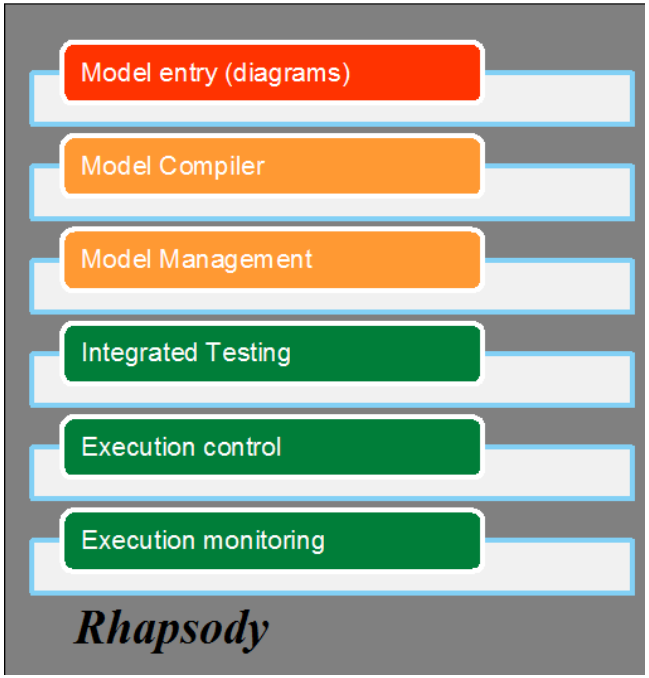


Figure 1: Rhapsody conceptual model

Rhapsody's graphical editor provides diagrams and tables to both enter and view model information. The model management portion of Rhapsody maintains the *model repository* – the information content of the model itself, and manages storage, recovery, and reporting. Beyond that, Rhapsody's *model compiler* constructs executable version of the model (provided that the model is *well-formed*) . The model compiler generates software source code to simulate the modeled system behaviors and properties. Rhapsody provides facilities to visualize the model execution – by showing state changes via dynamic coloring or by generating messages on sequence diagrams as model elements interact during the simulation.

Model execution control facilities give the engineer the ability to run, single-step, examine values, and set breakpoints. Additionally, web-based and panel-based views can be constructed to monitor and control the simulation. Beyond this, Rhapsody has a tool add-on called *Test Conductor* which supports the *UML Testing Profile*, and so can offer model-based testing specification, execution, verdicts, and management.

Rhapsody supports generation of code in a number of languages (notably, C, C++, Java, and Ada) and many compilers. In this book, we are generating code in C++ and will be using the popular Cygwin compiler. The Microsoft C++ compiler is also commonly used with Rhapsody as well and is almost completely compatible[1].

Rhapsody integrates with many other tools for special purposes. Notably, Rhapsody integrates with IBM DOORS and DOOR NG (Next Generation) for requirements traceability (although Rhapsody supports internal model traceability as well), many different version control tools (including Rational Team Concert), Simulink for control loop integration, SimulationX and Modelica for physics modeling and the Functional Mockup Interface (FMI) specification (http://fmi-standard.org).

[1] The only difference you're likely to notice is with the **cout** and **endl** applicators; in Cygwin you use them with the library context (as in "std::cout << "Hello " << std::endl;") while some versions of the Microsoft compiler wants you to move the library context ("cout << "Hello " << endl;").

4 Overview of the Harmony aMBSE Process

Harmony Agile Model-Based Systems Engineering (*Harmony aMBSE*) process focuses on the development of model-based system engineering work products such as requirements, architecture, interfaces, trade studies, and various analyses (such as safety, reliability, and security). It does this in an agile fashion by incorporating incremental development of engineering data, early and continuous verification of the correctness of that information, and continuous integration of the work of collaborating engineers. Figure 2 shows the overall process flow.

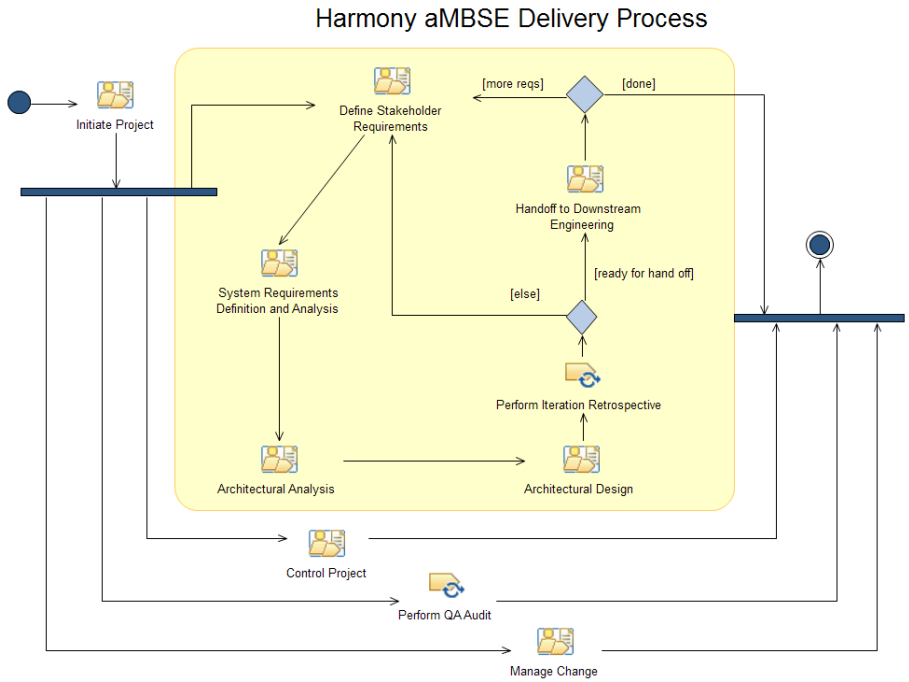


Figure 2: Harmony aMBSE Delivery Process

Each of the rectangular boxes in the figure represents a process activity, which, in turn, is defined by a set of nested activities or tasks. The diamonds represent decision points (at which only a single flow is taken at a time), while the horizontal bars are either forks or joins, which represent concurrent flows. The labeled pentagons are tasks on which one or more engineers work. Each task is defined with inputs and outputs, a purpose, description, the set of steps necessary to complete the task, and optional guidance material.

The activities and tasks of the Harmony aMBSE process shown in Figure 2 are[2]:

- Initialize Project
  Identify and prioritize stakeholder use cases, create the engineering team structure, create first cut schedule, risk management plan, and the System Engineering Management Plan.

- Define Stakeholder Requirements
  Identify stakeholders of interest, stakeholder needs as requirements, allocate these to use cases, and perform rudimentary requirements analysis, normally limited to scenario elaboration.

- **System Requirements Definition and Analysis**
  Identify system use cases (normally 1:1 match for the stakeholder use cases), derive system requirements, allocate them to use cases, analyze the use cases with computable models, create logical flow data and flow schema, analyze dependability, and create the initial system verification plan.

- **Architectural Analysis**
  Identify and analyze system trades and make technological and/or architectural choices based on that analysis

[2] Activities which are the focus of this Deskbook as in **bold**.

- **Architectural Design**
  Identify subsystems, allocate system requirements to subsystems, create subsystem requirements, create and allocate use cases to subsystems, update the logical data schema, develop control laws, and update dependability analyses.

- Control Project
  Perform project management activities, maintain risk management plan, and use daily meetings to enhance engineer collaboration

- Perform QA Audit (task)
  Perform quality assurance audits to ensure process compliance.

- Manage Change
  For work products under configuration management, control the change request process including review, assignment, resolution, and verification of each change request.

- Perform Iteration Retrospective (task)
  Ascertain the project's adherence to the project plan and look for opportunities to improve; also replan as necessary and appropriate.

- **Handoff to Downstream Engineering**
  Develop materials necessary for downstream engineering, including physical interface specification, creation of subsystem models, creation of a deployment (interdisciplinary) model, allocate requirements to engineering disciplines and define the interdisciplinary interfaces.

Let's look at the key activities in a little bit more detail.

4.1 Systems Requirements Definition and Analysis

This activity is a crucial one in the Harmony aMBSE process. In this activity, we will define the set of systems requirements (with traceability back to the stakeholder needs they will satisfy), group them into use cases, and then analyze them, a use case at a time, for completeness, accuracy, correctness, and consistency. We will do this through the development of a computable use case model and through this effort, we almost always find missing or incorrect requirements. In addition, we will do other work that uncovers other important requirements, such as modeling the logical data and flow schema (for things coming to or exiting from the system) and the system dependability (safety, reliability, and security) needs.

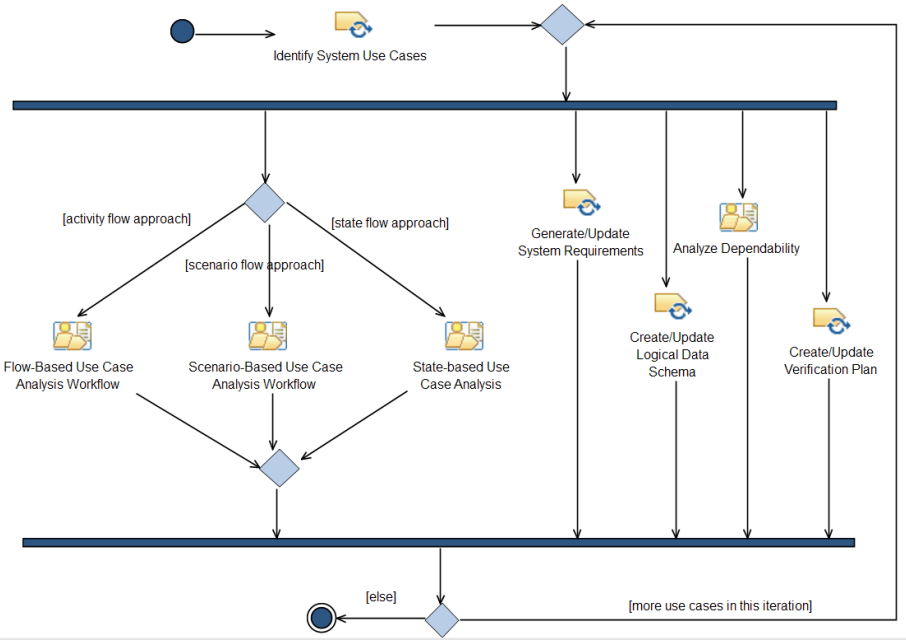


Figure 3: System Requirements Definition and Analysis

Figure 3 shows the overall workflow for this activity. Note that three different primary analytic approaches are supported – flow-based, scenario-based, and state-based. All accomplish the same purpose but using slightly different workflows.

In actual fact, there are five alternative workflows from which to choose. Figure 4 shows a decision tree for deciding which work flow to use. The options are:

- Flow-based
  This approach is best when the use case is heavily algorithmic, has

significant continuous flows, or is mostly flow based. This workflow is intended for use cases that are primarily focused on complex algorithms (such as encryption), continuous flows (such as fluid or energy), or when the flows into and out of the system predominate the use case behavior. In this case, executable models can be constructed with Rhapsody + Simulink, with fully executable activity diagrams, or poll-based state machines.

- Harmony "Classic"
  This is an older, less comprehensive approach and is deprecated, but still supported. This approach is only recommended for projects that have been started with the workflow defined in the previous version of Harmony SE but not for new development. Note that the "activity diagram" used here is not really a well-formed activity diagram but really is intended to be used as a summary of multiple scenarios. State machines form the normative black box behavioral specification.
- Activity-Based
  In this workflow, the primary purpose of the activity diagram is to identify system functions. This approach is recommended when the functionality of the system is less focused on input and outputs and more focused on the transformations the system performs. In this case, the work is aimed towards identifying and characterizing system functions. Similar to the "Classic" approach, activity diagrams here are used to summarize multiple scenarios rather than as a true model of behavior.
- Interaction-Based
  This option is best when working with non-technical stakeholders OR the use case is heavily interaction-based. This work flow is recommended when working with intended system users or other non-technical stakeholders, or when the interactions (as opposed to the system functions) are complex. The activity diagram is generally skipped in this workflow and the state machine forms the normative specification.
- State-based
  This approach is best when the use case is strongly modal or state-based AND you have strong expertise in developing state machines. This workflow is recommended for use cases that are either obviously state-based (such as automotive transmissions) or highly modal in nature. Note that this requires a generally higher level of technical skill on the part of the engineer.

While Figure 4 may look complicated, you will only be doing one of the five identified workflows for a given use case. It does provide options for different kinds of use cases, or when working with stakeholders or engineers with different skill sets. It should be noted that generally *any* of these workflows may be used for a use case analysis, so personal preference may be expressed as well. Note that each of these workflows involves the creation of an executable model – generally a state machine but it might also be an activity diagram.

Remember, all of these approaches work, so the selection of the best one is a matter of both personal preference and the nature of the problem being addressed. Later in this desk book we will use two of these approaches – flow-based and sequence-based – to illustrate the differences.

Functional Analysis of Requirements – Different Flows, Different Folks

Also notice the iterative nature of the workflows. Each has a "loop back" in the case of "more requirements." It is recommended that a small number of requirements be analyzed at a time so that the behavioral models are incrementally constructed. Experience has shown, time and time again, that delaying the analysis and execution of the behavioral model only serves to make the analysis much more difficult. The length of time for these feedback loops in the workflows should not exceed an hour or two; this is what we call the *nanocycle* and is key to the agility of the Harmony aMBSE process.

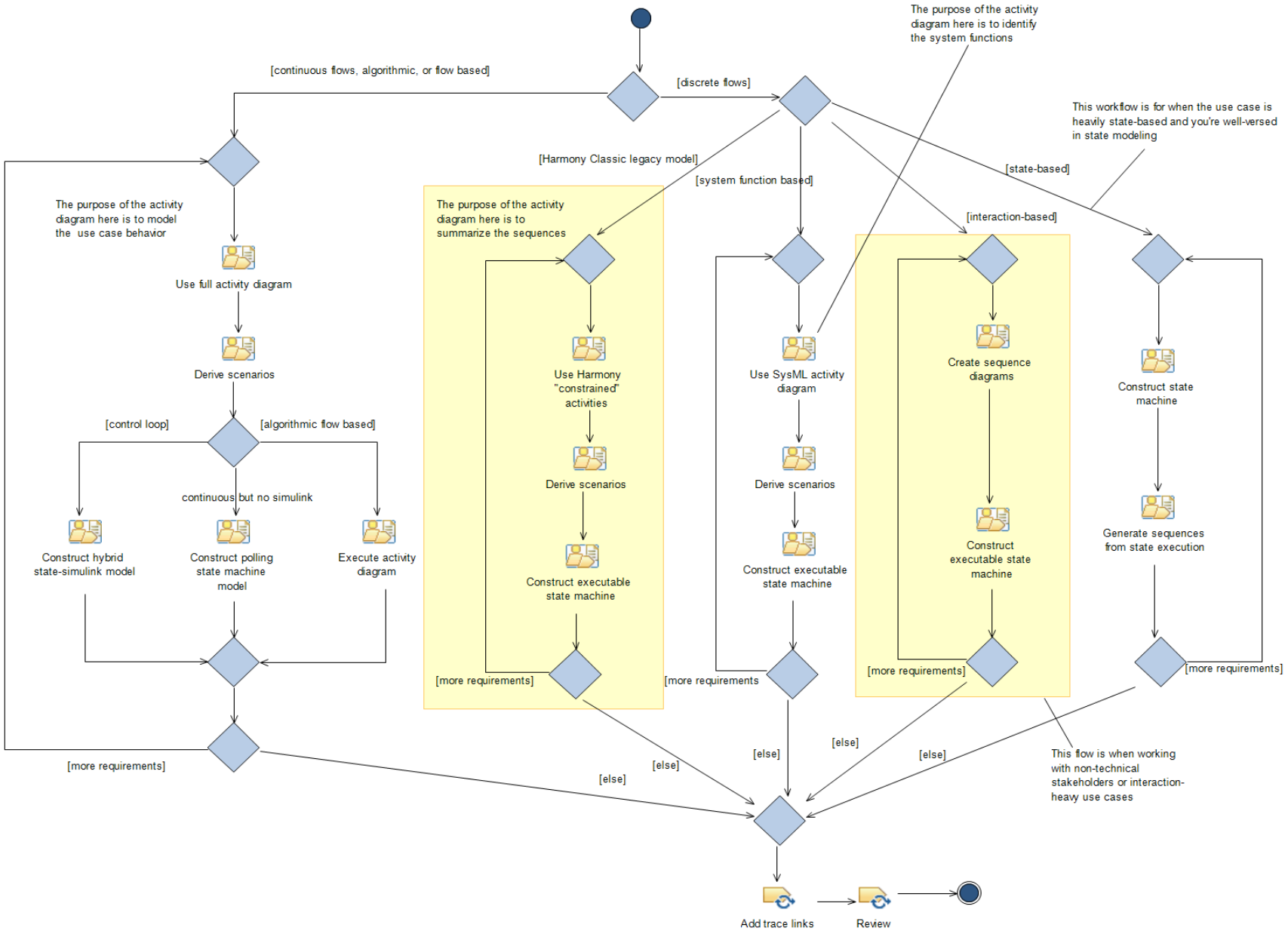


Figure 4: Use case analysis workflow decision tree

A Note about Use Cases

Use cases may be thought of as collections of system interactions and system functions around a common usage of a system. An alternative, but equivalent view is that they cluster requirements around a common system capability that involves interactions with elements in the system's environment.

Good use cases are independent from other use cases, at least in terms of requirements. This allows independent analysis of the use cases, allowing a "divide and conquer" strategy to address complex systems problems. Good use cases generally represent anywhere between 10 and 100 requirements and contain both functional and quality of service (QoS) requirements, such as performance, accuracy, fidelity, and reliability. This usually implies between three and 25 scenarios of interest, including both normal, or "sunny day" scenarios, and exceptional, or "rainy day", scenarios. We recommend incremental analysis of use cases, beginning with the sunny day scenarios and later adding in all the ways that things can go amiss.

There are two primary outcomes from the functional analysis of use cases. First, is the identification of problems with the stated requirements. In the course of analysis, it is very common to identify requirements that are missing, incomplete, inconsistent, or just plain wrong. As the analysis proceeds, requirements are fixed or added in parallel (see the *Generate/Update System Requirements* activity in Figure 3).

The second outcome is the identification and characterization of the logical interfaces. The term "logical" here means that we are defining the essential properties of the interfaces but not their ultimate realization. For example, we model most actor-system interactions as asynchronous events (which may or may not, carry data), but actual realization of these interfaces might be messages across a communications bus. It is important to note that incoming messages to the use cases invoke one or more system functions and messages to the actors are produced by one or more system functions. Different use cases often invoke common system functions and that is a point of potential co-dependence.

This leads us to the issue of "merging use cases." If use cases are independent, then merging use cases together in a larger scale analysis isn't difficult unless one of the following is true:

- Use cases are not completely independent in terms of requirements
- Use cases share system functions

The 2nd of these is the more likely. When the use cases are completely independent, then the actor-system interfaces are merely the sum of the messages from all the use cases that involve that actor. When they are not completely independent, the interfaces must be "merged" so that the common system functions are defined with a common definition: service name, inputs, outputs, pre-conditions, post-conditions, invariants, and definition of the required behavior of the system function.

Create Logical Data and Flow Schema

The purpose of this task is to characterize the flows into and out of the system. These flows may be discrete – such as in a commanded position to which to move the wing flap – or they may be continuous, such as the movement of water through a conduit. They may informational, such as the blood pressure of a patient undergoing a medical procedure; energy, such as the heat flow in a deicing system; materiel, as in a dispersal of projectiles; chemical, as in the diffusion of an anesthetic drug in a breathing circuit; fluid, as in the flow of air in a building heating system; or mechanical, as in the movement of a robot limb.

What all these flows have in common is their need to be characterized so that the quantities, statics, dynamics, and necessary precision of the system can be understood and so that good downstream engineering choices can be made. Typically, the metadata to be characterized includes topics such as:

- Set or range of acceptable values, including units
- The fidelity of control (Harmony aMBSE defines **fidelity** to be the "precision of an input to the system")

- The accuracy of control (Harmony aMBSE defines **accuracy** to be the "precision of an output from the system")
- Expected behavior if the data within range as well as out of range
- Safety impact of the flow
- Safety level of the flow (specific to the standard being used for conformance)
- Reliability of the delivery of the flow
- Security of the flow
- Whether the flow is measured, actuated (controlled), computed, or estimated
- Other "invariants" (assumptions)

The flow metadata is typically stored in *tags*, one of the means SysML provides for extension of the modeling language.

Again, the logical data and flow schema define the essential properties but not the physical realization of those data and flows.

Analyze Dependability

Dependability – literally "one's ability to depend upon a system" – has three primary aspects: safety, reliability, and security. These aspects are defined thusly:

- Safety is freedom from harm due to use, misuse or exposure to a system
- Reliability is a stochastic measure of the availability of services and flows
- Security is the ability of a system to resist attack

The first two of these aspects have a large and well-defined literature. Security for a cyber-physical system is less well defined but has been studied deeply in the information assurance field. In our systems context, Harmony has a broader scope of concepts and measures. Rhapsody has (optional) profiles available for the representation and analysis of these aspects of dependability. If you prefer to use other, specialized tools for this purpose, that's perfectly fine, as they are likely to have more capability than the Rhapsody profiles in those domains.

4.2 Architectural Analysis

Architectural analysis has a couple of applications. The first – on which we will not focus in this Deskbook – is to understand how the architecture behaves or performs under different circumstances. The second – which we will emphasize here – is to evaluate alternative technology and architecture choices against a set of acceptance criteria. The work flow for this use of architectural analysis in shown in Figure 5.

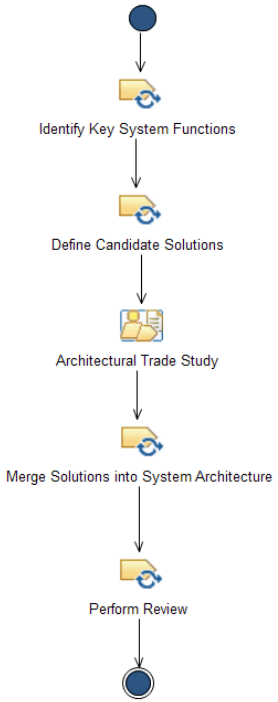


Figure 5: Architectural analysis workflow

Identify Key System Functions

The term "key functions" is a bit misleading. What it really means is to identify those system functions that can profit from optimization of technology or architecture choices.

Define Candidate Solutions

The candidate solutions are the technology or architecture choices that are reasonable solutions to meet the requirements. Technology choices might be to use a fluid-cooled versus an air-cooled system or a hydraulic, electronic, or pneumatic actuator. Architectural choices might be to use different architectural safety patterns for redundancy such as Triple Modular Redundancy or Heterogeneous Redundancy [4].

(Perform) Architectural Trade Study

The trade study itself has a nested workflow, shown in Figure 6.

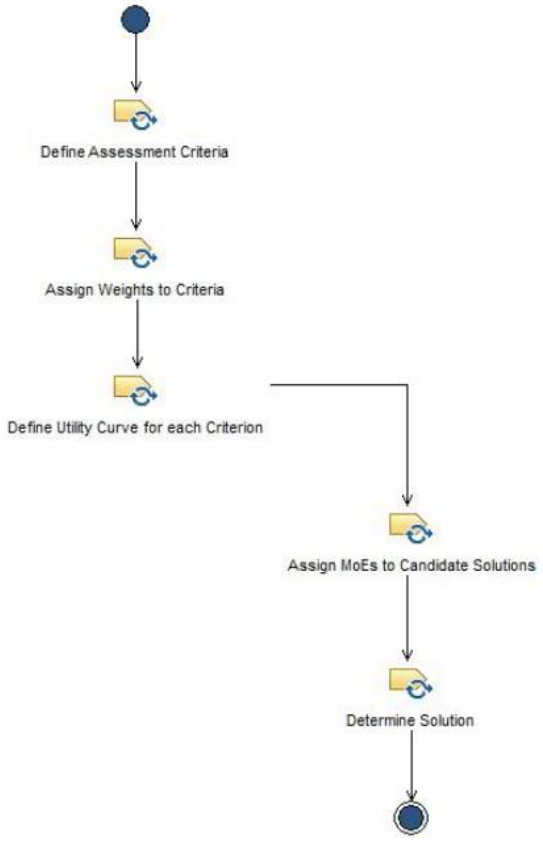


Figure 6: (Perform) Architectural trade study

Define Assessment Criteria

The assessment criteria are the aspects you want to optimize. Typical criteria might include:

- Recurring cost
- Development time
- Power required
- Reliability
- Safety
- Manufacturability
- Weight
- Performance
- Complexity
- Testability
- Accuracy
- Resource requirements (such as memory or computational power)

Assign Weights to Criteria

Not all criteria are equally important, so each criterion must be weighed with respect to its relative importance to the overall solution. This is often, but not necessarily done by normalizing the weights between 1 and 10.

Define Utility Curve for Each Criteria

The utility curve provides a means by which the different solutions may be evaluated as to how well that solution optimizes a specific criterion. A common technique is to construct a linear equation such that the worst solution under consideration results in a value of 0 and the best solution under consideration results in a value of 10; thus, most candidate solutions will be somewhere between 0 and 10.

Assign MOEs to Candidate Solutions

The assignment of the measures of effectiveness (MOEs) for each candidate solution is computed by applying the utility curve for each criterion to the solution and computing the weighted sum of the outputs of the utility curves.

Determine Solution

The selected solution is then the candidate solution which resulted in the highest MOE score among the evaluated candidates. Figure 7 shows a simple trades study in a table.

	Criteria and weights			
	Cost	Usability	Durability	
Candidate solution	**0.30**	**0.50**	**0.20**	**Candidate weighted score**
Touch screen	0.00	10.00	0.00	5.00
Membrane switches	8.00	7.86	1.00	6.53
Keyboard	10.00	0.00	10.00	5.00

Figure 7: Example trade study

Merge Solutions into Systems Architecture

The architecture is constructed by merging in the selected candidate solutions that emerge from the trade studies, in addition to other choices that were made without performing trade studies.

SysML and Rhapsody provide an additional means to do trade studies with parametric diagrams. With Rhapsody, you can define the equations in parametric constraints and then invoke third party mathematical tools, such as Maxima or Mathlab Symbolic Toolbox, to evaluate parametric diagrams. This is available in the Rhapsody Parametric Constraint Evaluation (PCE) Profile. Figure 8 shows an example parametric diagram.

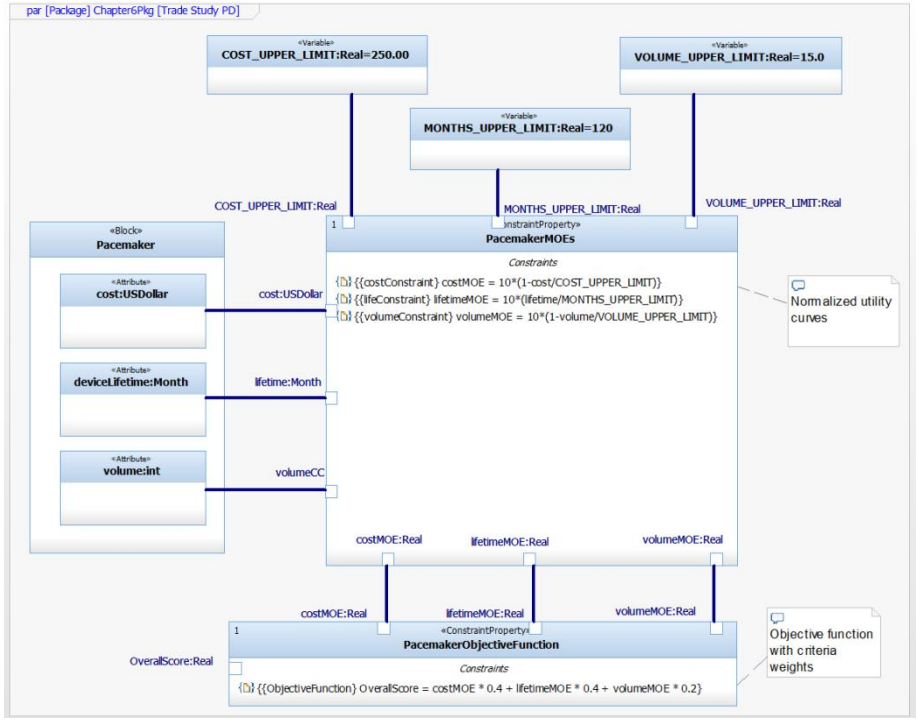


Figure 8: Example parametric diagram in Rhapsody

Evaluation of this parametric diagram for a candidate solution results in an output like that shown in Figure 9.

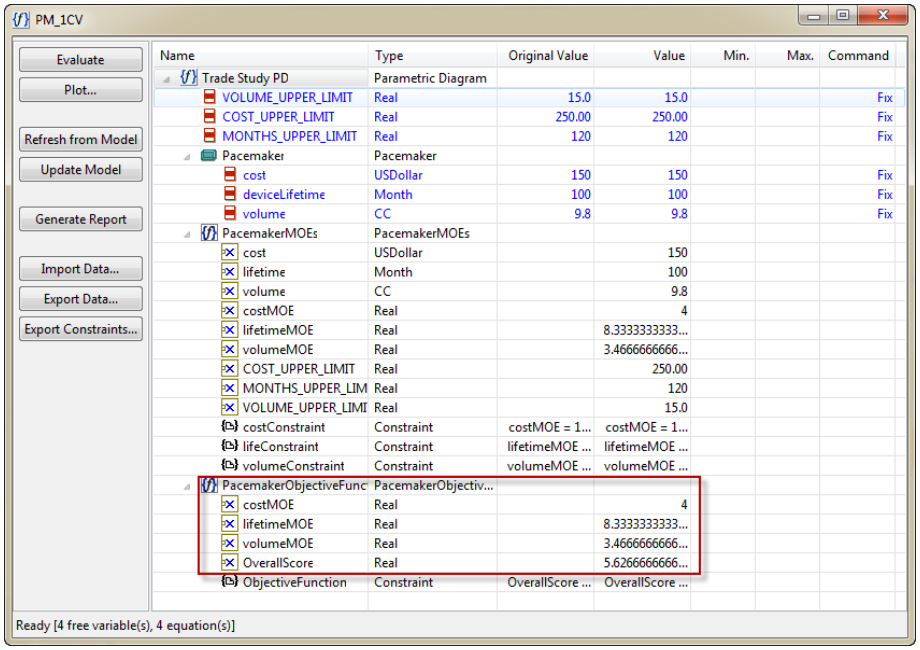


Figure 9: Example result from PCE evaluation

4.3 Architectural Design

The intent of architectural design in the Harmony aMBSE process is to

- Identify the subsystems
- Allocate requirements to subsystems
- Define the logical subsystem interfaces
- Update the data and flow schema
- Update the dependability analyses

A use case is almost never implemented by a single subsystem. This means that portions of a use case must be allocated to different subsystems. In practice, those portions are

- System requirements
- (Derived) subsystem requirements
- System functions
- (Derived) system functions
- (Derived) subsystem use cases

Figure 10 shows the Harmony aMBSE workflow for architectural design.

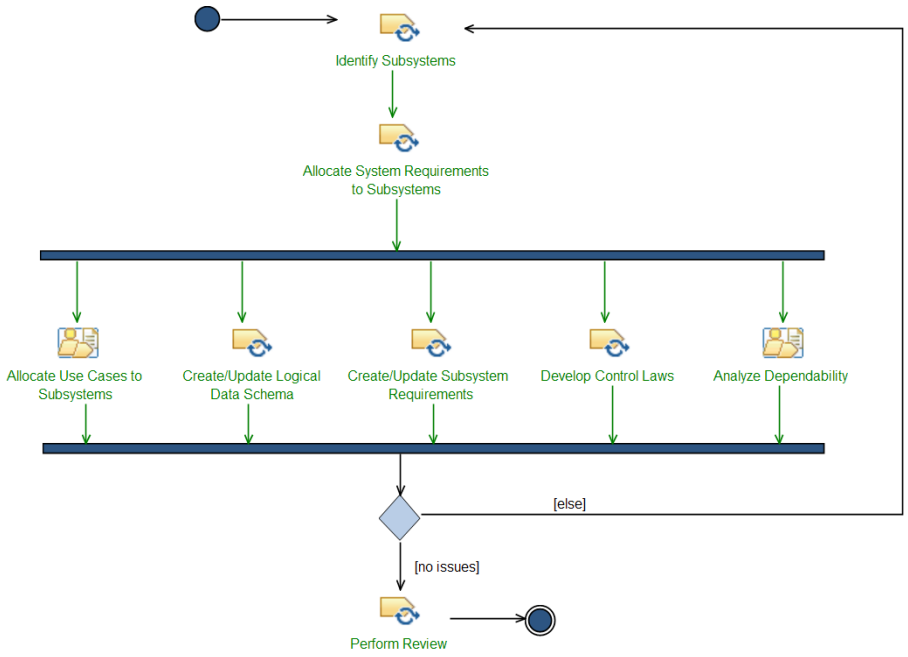


Figure 10: Architectural Design Workflow

Identify Subsystem

The subsystems are uses of blocks which represent the largest scale of system decomposition. Subsystems are generally implemented in terms of multiple engineering disciplines (e.g. software, electronics, mechanical, hydraulic, and pneumatic) by a single team. These subsystem teams perform what is collectively called *downstream engineering* in post-systems engineering activities, including software, electronic, and mechanical design.

One of the primary purposes of identifying these subsystems is to provide specifications for each subsystem team to follow. For this reason, the recommended model organization schema creates separate subsystem

packages to hold the relevant specifications (to facilitate the hand off). Information shared among subsystems is put into a common shared model.

The set of subsystem is shown on either (or both) block definition diagrams or internal block diagrams.

Allocation System Requirements to Subsystems

Some system requirements can be directly allocated to a single subsystem. However, many – if not most – must be decomposed into *derived requirements* which are then allocated. The decomposition is best done on requirements diagrams. Allocation relations (drawn from the subsystem to the requirement) may be done in either requirements diagram or matrices constructed for that purpose. They are best summarized in the matrices regardless of how they are constructed.

Allocate Use Cases to Subsystems

If only a few requirements are allocated to a subsystem, then they need not be allocated to subsystem-level use cases. However, many subsystems are themselves quite complex. Such subsystems can profit from exactly the same kind of analysis that we did for use cases at the system level.

There are two approaches to developing such use cases, as shown in Figure 11. The first – a part of the (deprecated) Harmony Classic SE process – is called *bottom up* because it allocates individual system functions (or subsystem functions derived from these) to the subsystems and then uses these as elements from which subsystem use cases may be constructed. The other approach, *top-down*, decomposes system use cases into subsystem-level uses cases via the «include» relation. In practice, smaller, less complex subsystem use cases are more easily developed with the bottom-up approach, but more complex use cases are better developed with the top-down method. In general, either approach may be used.

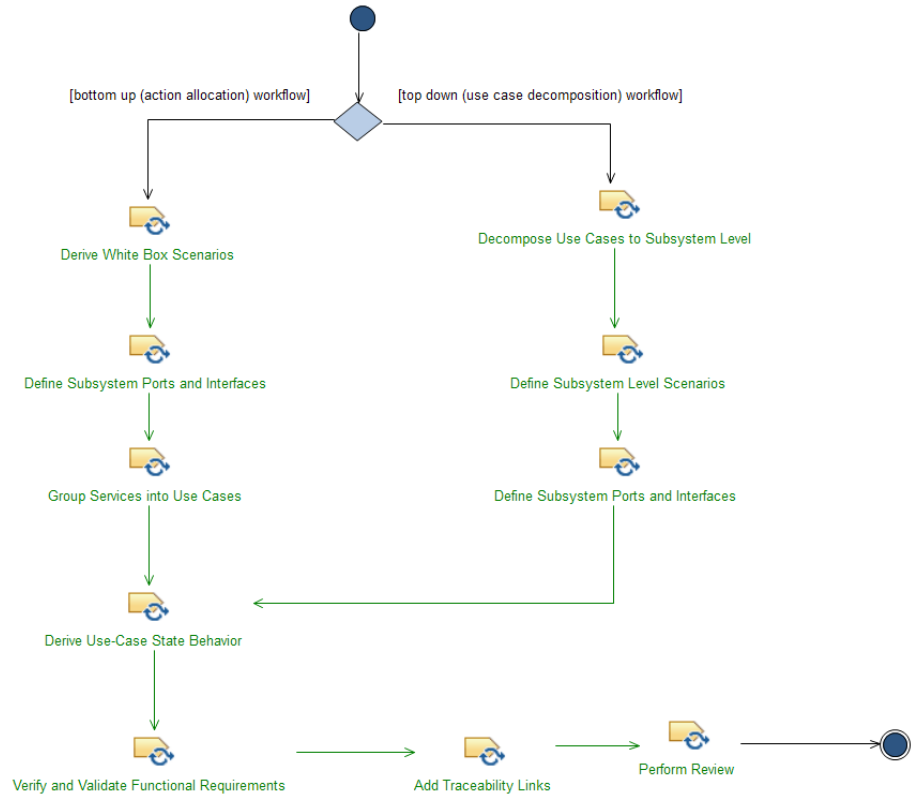


Figure 11: Allocate use cases to subsystems workflow

Create/Update Logical Data and Flow Schema

As we develop the logical subsystem architecture, many more data and flows are identified. They must be added to the data and flow schema.

Create/Update Subsystem Requirements

Just as we did for system requirements in the *System Requirements Definition and Analysis* activity, we need to repeat the activity to manage the subsystem requirements resulting from both the derivation from system requirements and from the analysis of the subsystem use cases.

Develop Control Laws

Control laws are most commonly expressed as proportional–integral–derivative relations representing closed-loop feedback control mechanisms. Most such control laws fit within a single subsystem and are, as such, out of scope here. However, some control laws are distributed between subsystems and these must be characterized as they affect the subsystem functions and interfaces. These may be defined as sets of partial differential equations or on control loop diagrams [6], most often using specialist tools, such as Simulink.

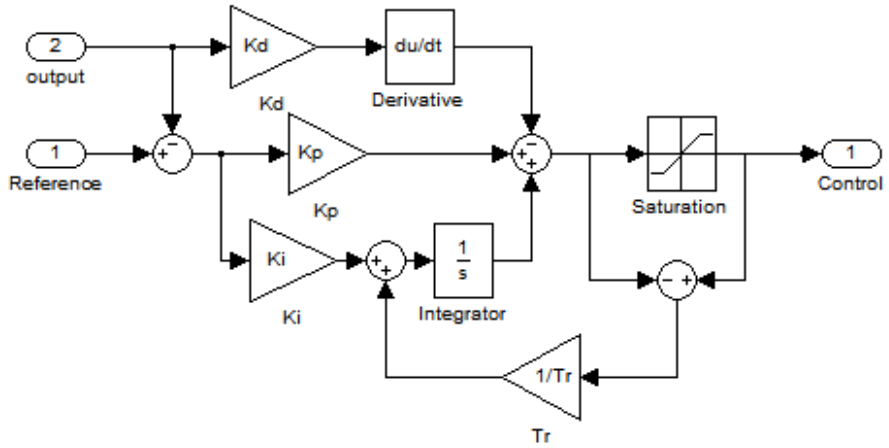


Figure 12: Example Control Loop Diagram

Analyze Dependability

The management of system dependability is an activity that goes on throughout the systems engineering process. Whenver engineers make technical, design, or implementation decisions, those decisions must be evaluated for their impact on system safety, reliability and security. Most commonly, such analyses identify the need for new requirements to account for dependability concerns introduced with technical decisions.

4.4 Handoff to Downstream Engineering

Once the subsystem and interface specifications are ready, they must be handed off to the subsystem teams for the performance of downstream engineering activities. This involves two primary (sub)workflows. Firstly, the physical interfaces and physical data and flow schema must be derived from their logical counterparts. In the Harmony aMBSE process, we recommend this is put into a separate *shared model* for inclusion (by reference) into all subsystem models[3]. Secondly, a separate model must be created for each subsystem and populated with its specification from the systems engineering model. Also, a *deployment architecture* must be created for each subsystem. This deployment architecture identifies the engineering disciplines involved in the design and implementation of the subsystem, the (derivation and) allocation of requirements to those participating disciplines, and specifies the interfaces between the engineering disciplines. This readying of the subsystem model requires the participation of engineers from each supporting discipline as well as one or more system engineers. The handoff workflow is shown in Figure 13.

[3] See Chapter 10 for more information on model organization.

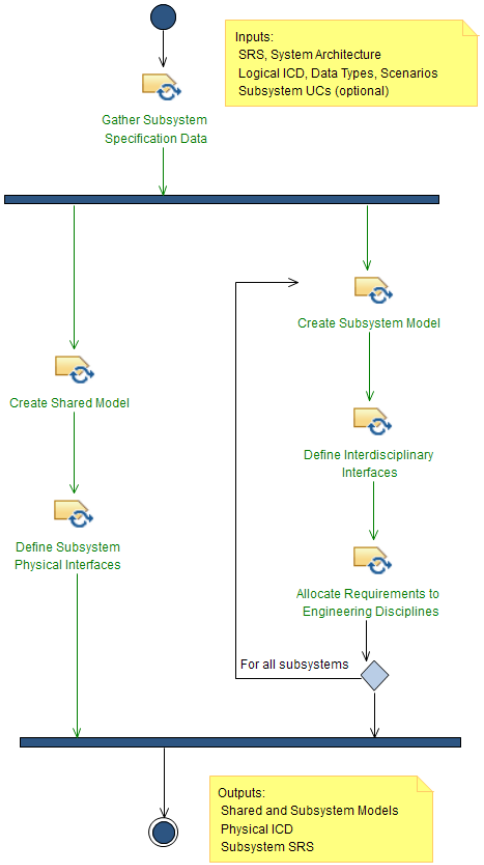


Figure 13: Handoff to downstream engineering workflow

5 The Harmony SE Toolkit

This document details the features and functions of the *Systems Engineering Toolkit* shipped with Rhapsody version 8.2.1. If you have an earlier version then most of the document will still apply, however some functions may be different.

The Systems Engineering Toolkit (referred to from here on as the SE Toolkit) is installed automatically as part of the Harmony-SE profile and contains a wealth of useful features for automating the building and checking of systems engineering models.

All SE Toolkit features (except for the startup wizard) are invoked from the contextual (right-click) menu of model elements in the browser, on a model element, or a diagram itself.

All SE Toolkit features are found under the SE-Toolkit menu:

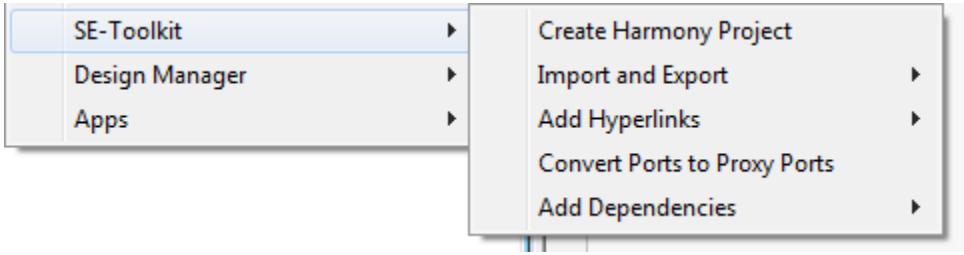


Figure 14 SE-Toolkit menu

Just a few of the most important helpers are described in this section. For a full description of all of the SE-Toolkit functionality, see the **Systems Engineering Toolkit Handbook**, available for download at Merlin's Cave (http://merlinscave.info/Merlins_Cave/Tutorials/Entries/2017/2/7_SE-Toolkit_Handbook.html).

5.1 The Harmony-SE Profile

The Harmony-SE profile loads the Systems Engineering Toolkit. It also contains new terms used in the Harmony workflow, along with stereotypes and tag values that allow user-customization of the SE Toolkit features. The profile also contains some custom table and matrix layouts.

In addition the profile loads in property files (.prp files) which override Rhapsody's default properties. These property files are loaded hierarchically as shown below:

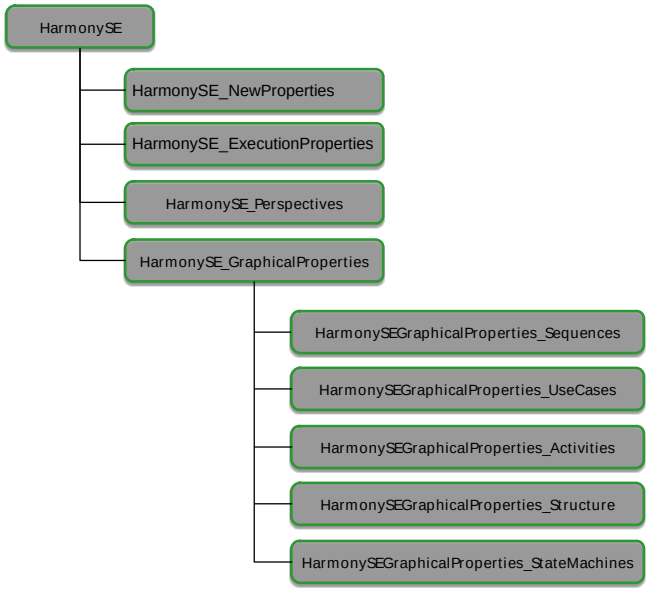


Figure 15 Harmony SE Property Files

5.2 Functional Analysis Helpers

5.2.1 Import Description from RTF

5.2.1.1 Intent

Import an existing RTF file as the description for a selected model element – either as the finished description or as a 'template' – that is a partially filled description.

5.2.1.2 Invocation

The helper may be invoked from the context menu of:

- The Project
- Use Cases
- Blocks
- Operations

Menu Entry: SE Toolkit → Import Description from RTF

5.2.1.3 Basic Operation

When invoked, the tool will look for a tag called `descriptionTemplate` – starting on the current element and then looking 'up the tree'. The first `descriptionTemplate` tag found in this way is used – this allows more than one template to be used for different areas of the model. Note that only the model has this tag out of the box as described below. If you wish to use different templates for different parts of the model, then this tag must be manually added (for example adding a new tag to a Use Case will cause only that Use Case to use the template – adding the tag to a Package will allow all elements in that Package to use the same template)

The tag should contain the full path to an RTF file. The path may be a fixed one or may contain the following keywords:

- {OMROOT} – will be replaced with the Rhapsody root directory
- {PROJECT} – will be replaced with the current project directory
- {PROJECT_RPY} – will be replaced by the _rpy folder for the current project (useful for controlled files which by default are stored there)
- {TYPE} – will be replaced by the user defined metaclass of the selected model element (for example use Block – not Class). Note that this must match the type exactly – for example use "UseCase" – not "Use Case"

Note that the profile contains a stereotype, which if applied to project, adds a project level tag `descriptionTemplate` with the default value:

{OMROOT}/Profiles/HarmonySE/{TYPE}

The profile also contains several example RTF templates (as controlled files):

- UseCase template
- Block template
- Operation template

5.2.2 Create System Context

5.2.2.1 Intent

Create a system context diagram from the Actors which associate with a Block.

5.2.2.2 Invocation

The helper may be invoked from the context menu of a Block.

Menu Entry: SE Toolkit → Architecture Tools → Create System Context

5.2.2.3 Basic Operation

When invoked the tool will create Actor Blocks for each connected Actor. A part is created for each Actor Block and the original System Block. Ports and interfaces are created between these elements and everything is placed onto an Internal Block Diagram. Note that all created artifacts are placed into the same Package as the original Block, except for Interfaces which are created in the Interfaces Package.

For example:

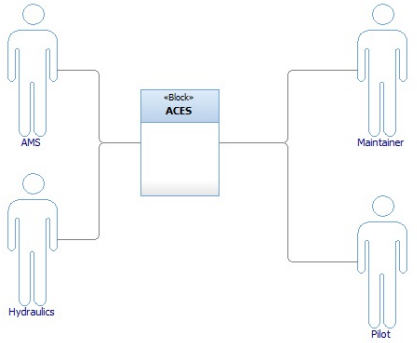


Figure 16 Actors connected to a System Block

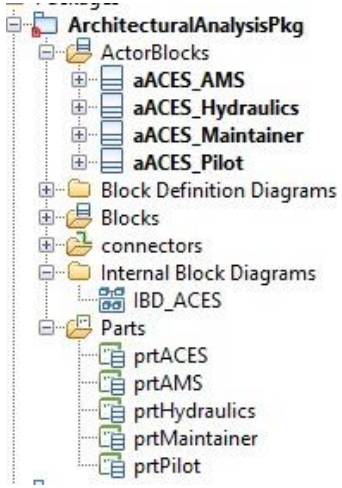


Figure 17 Created Actor Blocks, Parts, Connectors etc.

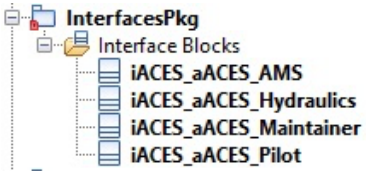


Figure 18 Created Interfaces

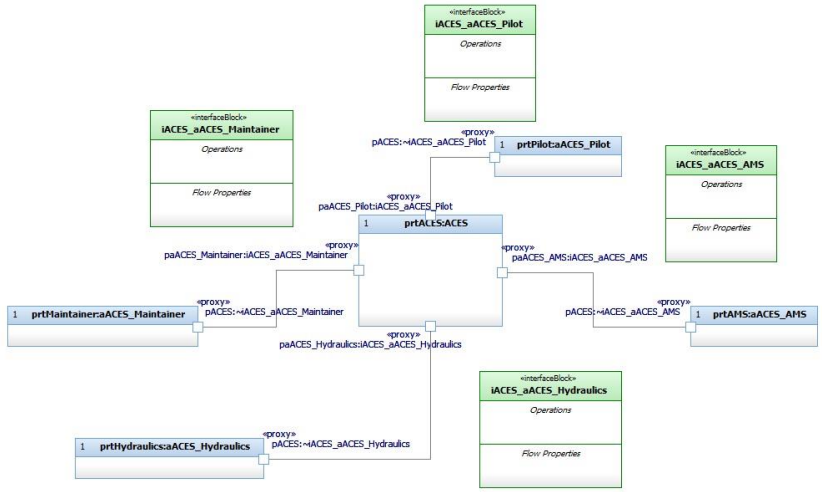


Figure 19 Created System Context Diagram

5.2.3 Create System Model from Use Case

5.2.3.1 Intent

Create a use case functional model for the selected use case for the purpose of constructing a computable model of the use case.

5.2.3.2 Invocation

The helper is invoked from a Use Case.

Menu Entry: SE Toolkit → Create System Model From Use Case

5.2.3.3 Dependencies

5.2.3.3.1 Location of Use Case Model

To create the functional use case model, the helper needs to know where to create the new model elements. By default, it looks for a package called FunctionalAnalysisPkg. If this package is not found, then the helper cannot continue. See the customization section for information on how to change this.

5.2.3.4 Basic Operation

5.2.3.4.1 Execution Considerations

The helper assumes that at some point the use case functional model will be executed. Any artifact that executes must have a formal name – that is a name with no spaces or special characters. To that end before creating model elements, the helper checks the use case name and creates a corresponding executable name (removing spaces and special characters). Artifacts created in the use case functional model use this executable name rather than the original use case name.

5.2.3.4.2 Use Case Model - Classic

The general form of the use case model created by the helper is shown below (the original use case is shown in orange)

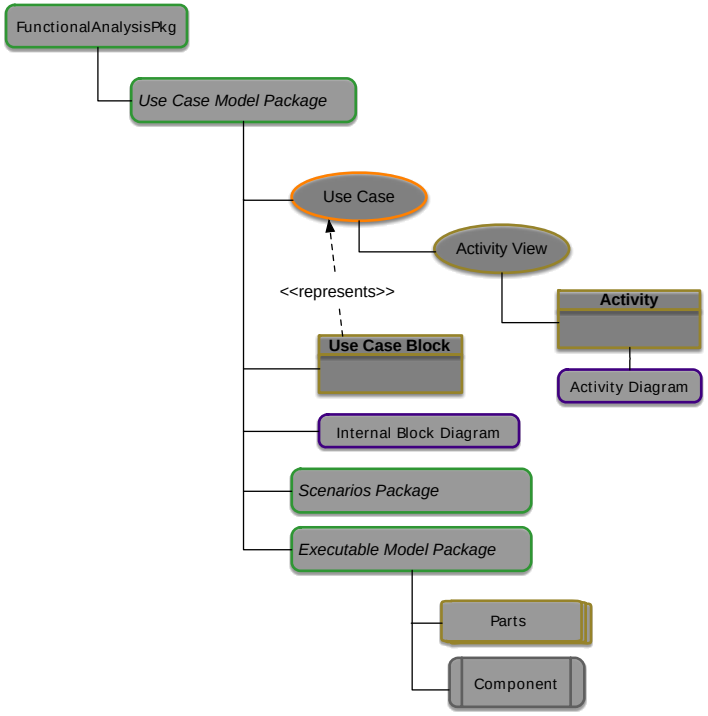
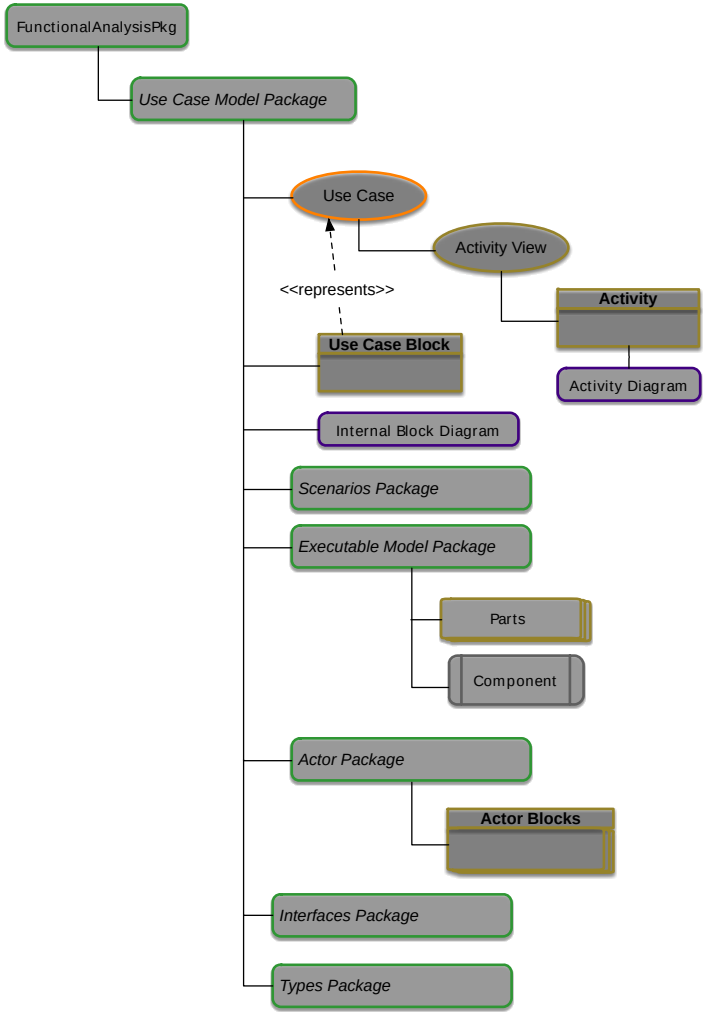


Figure 20 Use case model

5.2.3.4.3 Use Case Model - Agile

In Agile mode, each use case model also has its own types package, interfaces package and actors package (these are options controlled by properties). An use case-specific Actor Block is created for each connected actor, the block has a <<represents>> dependency back to the original Actor.



5.2.3.4.4 Package Structure

The helper creates the following package structure, where *UCName* is replaced with the executable name as described above.

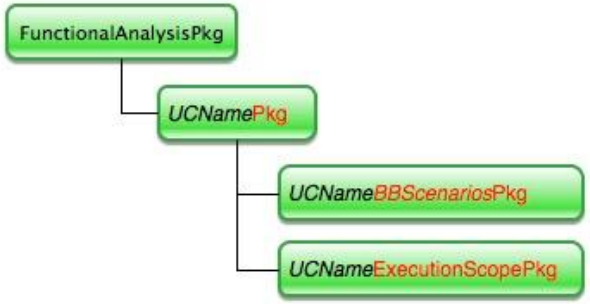


Figure 21 Use case functional model package structure

5.2.3.4.5 Use Case Relocation

After the creation of the use case it is at first located in the **RequirementsAnalysisPkg**. With the creation of system model from use case it is moved into the new use case functional model package (*UCName*Pkg).

Relocation of the Use Case is an option controlled by the following property:

```
SEToolkit.CreateSystemModelFromUseCase.MoveUseCase
```

5.2.3.4.6 Actor Relocation

Each Actor connected to the use case is moved into the *ActorPkg*. If the *ActorPkg* does not exist, the Actors are left where they are. Relocation of Actors is an option controlled by the following property:

```
SEToolkit.CreateSystemModelFromUseCase.MoveActors
```

See the Customization section for details on how to change where the Actors are relocated.

5.2.3.4.7 Internal Block Diagram

A new internal block diagram is created with the name IBD_UCName. The IBD is populated with the parts of the Use Case Block and the associated Actors or in case of agile the Actor Blocks.

5.2.3.4.8 Use Case Block

A block is created to represent the Use Case, named UC_*UCName.* The Block receives a dependency to the Use Case stereotyped <<represents>>

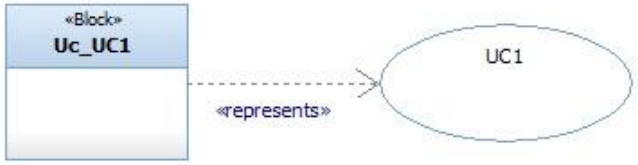


Figure 22 Use Case Block Dependency

5.2.3.4.9 Actor Blocks

In agile mode – an ActorBlock is created for each associated Actor and placed into a new Package (with the name *UCName*ActorPkg). This behavior is controlled by the following property:

```
SEToolkit.CreateSystemModelFromUseCase.CreateBlocksFromActors
```

ActorBlocks are named a*UCName_ActorName* and are given a <<represents>> dependency back to the original Actor:

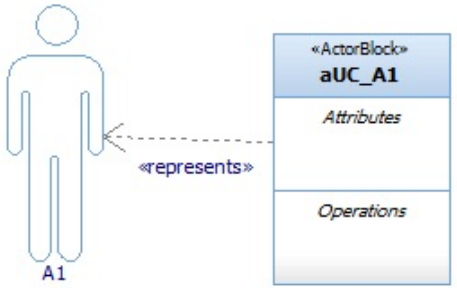


Figure 23 Actor Block

Note that Actor Block names use an abbreviated form of the Use Case name, using only the uppercase characters. For example, an Actor called *Driver* connected to a Use Case *Operate Vehicle* would result in an Actor Block called aOV_Driver. This behavior is controlled by the following property and is on by default in agile mode:

SEToolkit.CreateSystemModelFromUseCase.
AbbreviateActorBlockName

In this example if the option is switched off the Actor Block name would instead be *aOperateVehicle_Driver*.

Use Case inheritance is also supported – that is, if one use case specializes another, the more specialized use case will inherit any actor associations of the more general use case. This is an option controlled by the following property:

SEToolkit.CreateSystemModelFromUseCase.UseInheritedUseCase
Actors

5.2.3.4.10 Executable Use Case Model

An instance of the use case block is created (that is a part typed by the use case block). A part is created for each actor (or in agile mode each actor block) connected to the use case. These artifacts are placed in the UCNameExecutionScopePkg and are also placed on the internal block diagram.

5.2.3.4.11 Activity View

A new activity view is created under the use case. Since these do not execute (instead they are intended to model the functional flow) they simply take the name of the use case and add the suffix *Black Box View.* A new activity and activity diagram are created under this activity view.

5.2.3.4.12 Dependencies

A dependency is added from the activity to the use case block, stereotyped <<SDGenerationTarget>>. This is to allow the sequence diagram generator helper to automatically detect the appropriate lifeline to use when generating black box sequence diagrams.

A dependency is added from the activity to the BBScenariosPkg, stereotyped <<scenarios>>. This is to allow the sequence diagram generator helper to automatically select the package in which to place generated sequence diagrams.

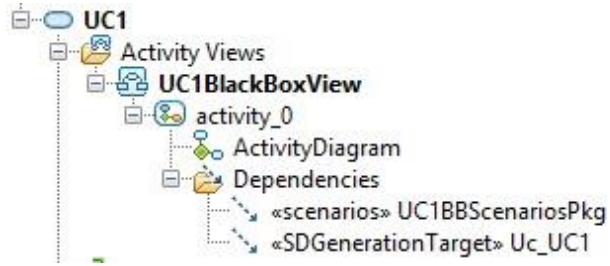


Figure 24 Activity dependencies

5.2.3.4.13 Ports and Interfaces

In agile mode, ports and interfaces are created for the Use Case and Actor Blocks. In addition, links are created between the parts and those are also shown on the Internal Block Diagram. Note that these are of course empty at this point – they will be later populated through scenario analysis. This behavior is controlled by the following property and is on by default in agile mode:

SEToolkit.CreateSystemModelFromUseCase.CreatePortsAndInter
facesWithSystemModel

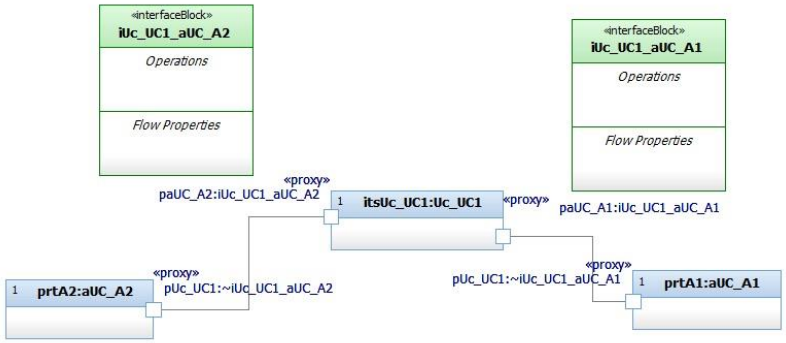


Figure 25 Ports and Interfaces on IBD

5.2.3.4.14 Interfaces and Types Packages

In agile mode two additional packages are created as part of the system model – a types package called *UCName*TypesPkg and an interfaces package called *UCName*InterfacesPkg. This behavior is controlled by the following properties which are both switched on by default in agile mode:

SEToolkit.CreateSystemModelFromUseCase.CreateLocalTypesPackage

SEToolkit.CreateSystemModelFromUseCase.CreateLocalInterfacesPackage

5.2.3.4.15 Hyperlinks

For ease of later navigation, hyperlinks are added from the use case to the activity diagram and internal block diagram.

5.2.3.5 Example

For the example use case shown below:

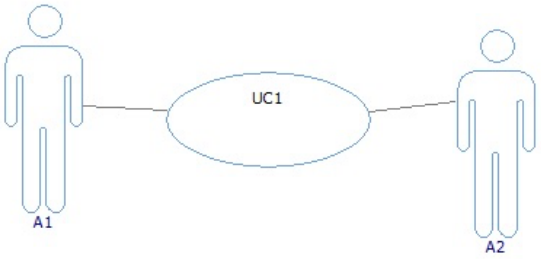


Figure 26 Example Use Case

The following use case model is created (agile mode) shown:

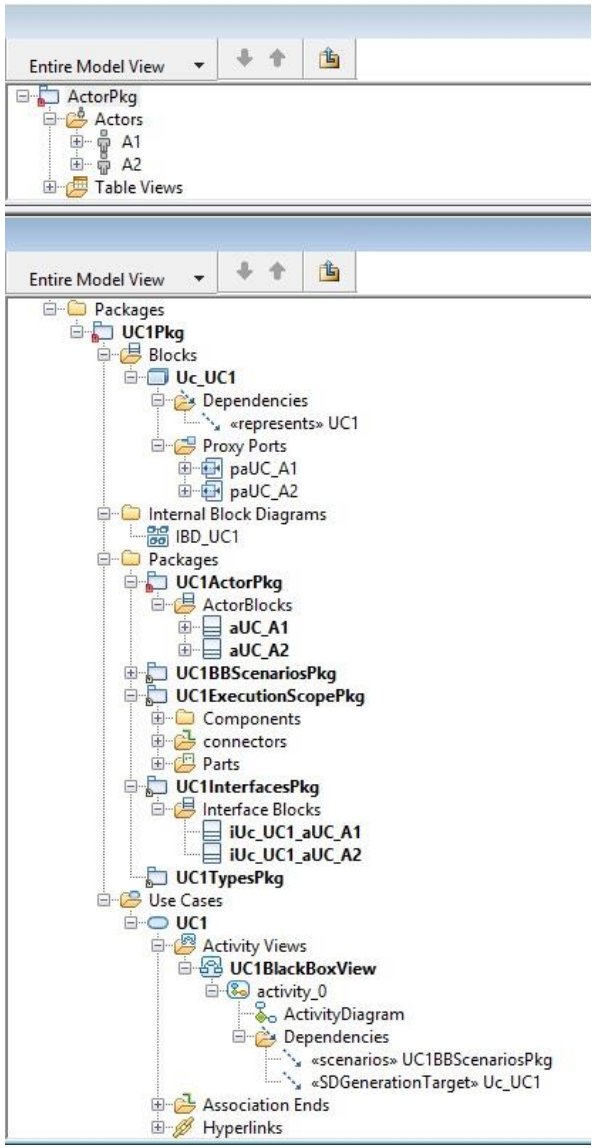


Figure 27 Example use case model

5.2.3.6 Customization Options

5.2.3.6.1 Functional Analysis Package

When *Create System Model from Use Case* executes – it requires a 'root' package in which to create new artifacts – by default that is a Package called FunctionalAnalysisPkg. The name and location of the "Functional Analysis Package" to use may be modified in two ways – locally or globally.

5.2.3.6.2 Modifying the Functional Analysis Package Locally

The tool looks 'up the tree' from the currently selected element, looking for a named package. The property that controls that name is a regular expression:

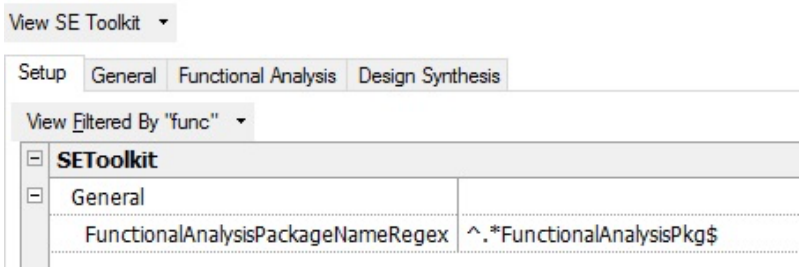


Figure 28 Functional Analysis Package Name Regular Expression

What this means is that by default – the first Package found whose name ends in "FunctionalAnalysisPkg" will be used. By modifying the regular expression, you could change the naming strategy used.

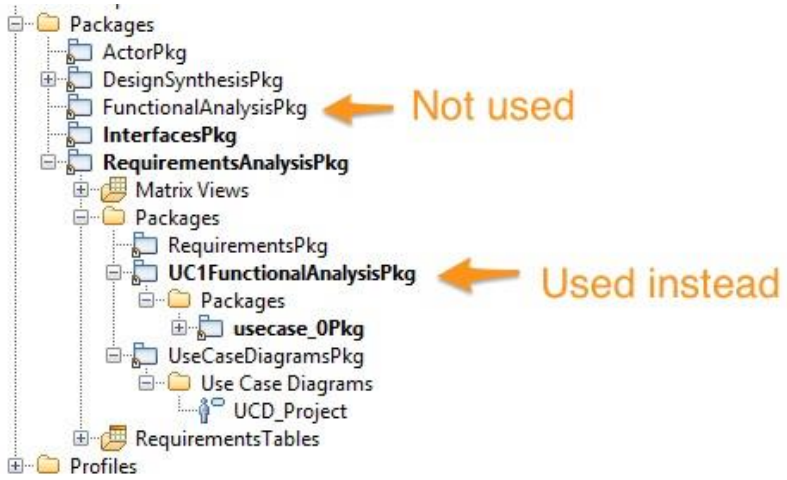


Figure 29 Example of a Local Functional Analysis Package

5.2.3.6.3 Modifying the Functional Analysis Package Globally

To make a more global change, apply the 'HarmonySE' stereotype to the Project – this adds a tag to the project: `FunctionalAnalysisPkg` – of type Package. A different "Functional Analysis Package" may then be specified by modifying the value of the tag (regardless of the actual name of the Package to be used)

Note that if you have already created a model for a use case, setting this tag will result in duplicate artifacts – a new use case model will be created for the use case regardless of whether one already exists in another location.

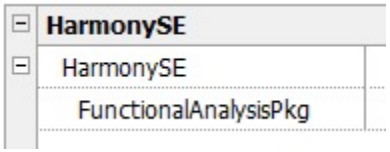


Figure 30 Changing the root package for use case models

5.2.3.6.4 Actor Package

The tool also attempts to relocate Actors. The selection of which package to move the Actors into is performed in the exact same way as described above for the functional analysis package – there is a corresponding

property to look for a named package and a tag on the project level stereotype to specify a global one.

5.2.4 Create Scenario ("Generate Sequence Diagram")

5.2.4.1 Intent

To create a basic sequence diagram with an initial set of lifelines to allow scenario modeling to proceed consistently.

5.2.4.2 Invocation

This helper may be invoked from an Activity View or a Use Case. (Note that if you are following either of the Harmony workflows you should not activate this tool on a Use Case – but on the Activity View instead). See the Interaction-based workflow on Figure 4

Menu Entry: SE Toolkit → Create Scenario

5.2.4.3 Basic Operation

The tool creates a new sequence diagram (with a default name) in the BBScenariosPkg. It detects all associated parts and adds them to the diagram as lifelines. These lifelines are set to show their label rather than their name for readability. This is an option controlled by the following property:

`SEToolkit.CreateScenario.UseLabelsOnLifelines`

Additionally, the HarmonySE Profile contains a Comment called SDDescriptionTemplate. A copy of this comment is made (owned by the Sequence Diagram) and is placed on the diagram (lifelines are shifted over to accommodate it). This is an option controlled by the following property:

`SEToolkit.CreateScenario.AddCommentToScenario`

The created Sequence Diagram is also added as a reference to the Activity View to allow for later consistency checking.

Example:
For the following Use Case Diagram:

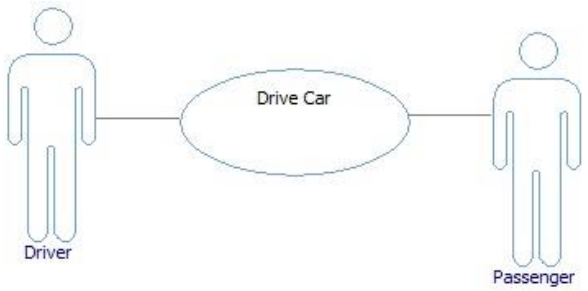


Figure 31 Create Scenario - Use Case Diagram

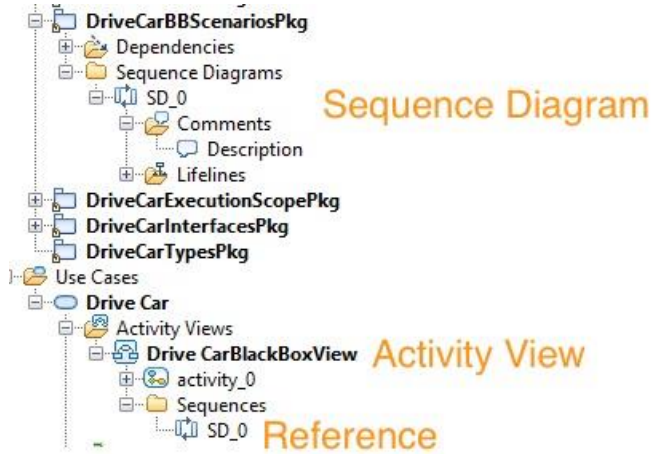


Figure 32 Create Scenario - Created Artifacts

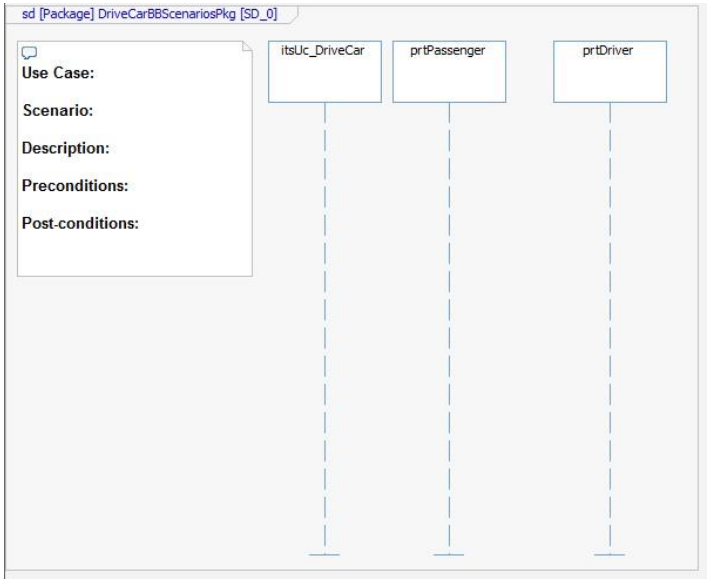


Figure 33 Create Scenario - Created Sequence Diagram

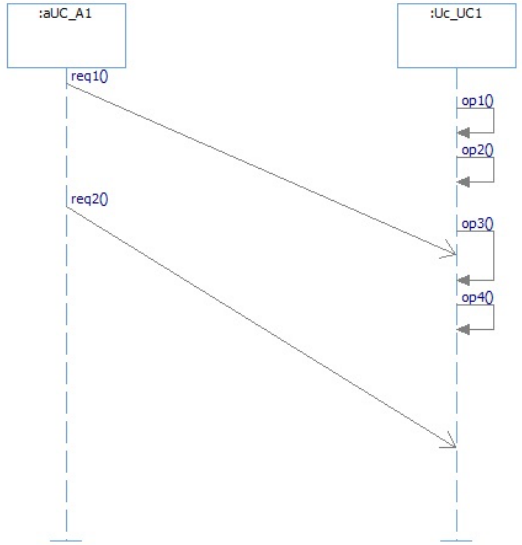


Figure 34 Straighten Messages - Before Invocation

5.3 Miscellaneous Helpers

5.3.1 Straighten Messages

5.3.1.1 Intent

When animating a model, Sequence Diagrams may be generated from the animation. Such diagrams show events as slanted lines – indicating that they are received some time after they are sent. This representation can make the diagrams needlessly long and less readable. This helper straightens such messages to aid readability.

5.3.1.2 Invocation

This helper may be invoked from a Sequence Diagram.

Menu Entry: SE Toolkit → Straighten Messages

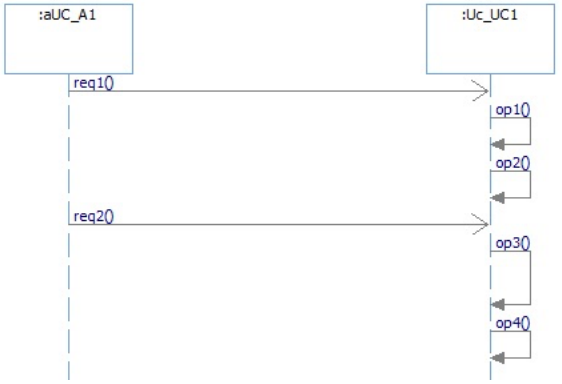


Figure 35 Straighten Messages - After Invocation

5.3.1.3 Basic Operation

Any messages that are not already horizontal will be made so. For example:

5.4 Summary

Rhapsody provides the Harmony SE Profile and the SE Toolkit to provide some automation of common system modeling tasks. It is important to remember:

> The toolkit provides absolutely no functionality that a competent engineer cannot perform themselves with a small amount of effort. In some cases, the output of the toolkit is intended to provide a starting point that will be elaborated and embellished by the systems engineer.

Figure 36 below summarizes the capabilities of the SE Toolkit.

SE Toolkit Feature	Description	Primary use in Harmony aMBSE Process
Add Hyperlinks	Adds a hyperlink from the source(s) to the destination(s)	Generic
Add Dependencies	Adds a dependency from the source(s) to the destination(s) with the specified stereotype	Add Traceability Links
Add Referenced Sequence Diagrams	Adds sequence diagram(s) as referenced sequence diagrams to the selected use case or activity view	System Requirements Definition and Analysis Allocate Use Cases to Subsystems
Add as Reference	Adds selected sequence diagram(s) as referenced sequence diagrams to a use case or activity view	System Requirements Definition and Analysis Allocate Use Cases to Subsystems
Convert Ports to Proxy Ports	Converts ports to standard ports and interfaces to interface blocks	Generic
Convert Port to Proxy Port	As above but for a single selected port	
Show Startup Wizard	Shows the harmony startup wizard which allows the user to set default harmony-related properties	
Refactor Action Name	Allows an action to be "renamed" – the tool refactors any other actions in this – or sub/referenced activities so they have the same action statement	
Select Sequence Diagram to Reference	Maps an interaction occurrence on a sequence diagram to a sequence diagram	
Merge Block Features	Copies operations, receptions, and values from the source blocks to a single destination block	Generic
Straighten Messages	Cleans up an animated sequence diagram	Generic
Duplicate Activity View	Creates a duplicate of the selected activity view – removing any referenced sequence diagrams	Architectural Design
Create Test Bench	Creates a test bench statechart on an actor by analyzing the actors ports	Generic
Allocation Operations from Swim Lanes	Copies operations allocated to a swim lane in an activity diagram to the relevant subsystem blocks	Allocate Use Cases to Subsystems (Bottom-up approach)
Generate Allocation Table	Creates a table (csv file) of the allocation decisions made on an activity diagram and adds to the model as a controlled file	Allocate Use Cases to Subsystems
Generate	Creates a sequence diagram	System Requirements

Sequence Diagrams	by processing object and/or control flows on an activity diagram	Definition and Analysis
Browse References	Provides an enhanced references browser	Model exploration
Create Harmony Project	Creates a Harmony project model structure	Project Initialization
Create System Model from Use Case	Creates a block context model in a compliant package structure from a use case	System Requirements Definition and Analysis Allocate Use Cases to Subsystems
Create Call Behavior	Creates a new activity and a call behavior for it from the selected action on an activity diagram	Generic
Auto Rename Actions	Harmonizes the action statement and action name in an activity diagram	System Requirements Definition and Analysis Allocate Use Cases to Subsystems
Add Actor Pins	Add Harmony-specific *actor pins* to activities on an activity diagram	System Requirements Definition and Analysis Allocate Use Cases to Subsystems
Perform Activity View Consistency Check	Checks the consistency between the actions on an activity diagram and the operations on a set of sequence diagrams	System Requirements Definition and Analysis Allocate Use Cases to Subsystems
Create Ports and Interfaces	Creates behavioral ports and associated interfaces (or proxy ports and associated interface blocks) based on the interactions on sequence diagrams	System Requirements Definition and Analysis Allocate Use Cases to Subsystems
Create Delegation Ports	Creates new delegation ports on the boundary of a system block to delegate messages to its internal parts	Architectural Design
Connect Ports	Creates links between ports on an Internal Block Diagram	Generic
Create Scenario	Creates a new sequence diagram from the selected use case or activity view by analyzing the connected actors	System Requirements Definition and Analysis
Merge Functional Analysis	Copies operations, receptions, and values from all use case blocks into a selected block	System Requirements Definition and Analysis Allocate Use Cases to Subsystems
Duplicate Activity View	Copies an activity diagram and strips away from the copy any referenced sequence diagrams	Generic
Create Subpackages	Creates a package per subsystem and moves subsystem blocks into those packages	Architectural Design
Allocation Wizard	Copies features (operations, receptions and attributes) from one architectural layer to another and tracks where features have been allocated	Architectural Design
Perform Allocation Consistency Check	Checks consistency between the allocation actions in swim lanes and the allocation operations in subsystem blocks	
Perform Activity View Consistency Check	Checks consistency between the actions in swim lanes and the operations on referenced sequence diagrams	Generic

Create Operations from Call Operations	Creates new operations in a block from 'empty' call operations on an activity diagram	Generic
Setup Model Execution	Creates an executable, animated, web-enabled component with the correct scope to execute a single use case model	System Requirements Definition and Analysis
Generate Allocation Table	Summaries the allocations of operations of a white box activity diagram into an Excel spreadsheet	Architectural Design
Generate N2 Matrix	Creates an Excel spreadsheet for the provided and required interfaces from an internal block diagram	Architectural Design
Copy MoEs to Children	Copies the MOE attributes of key function block into the solution blocks	Architectural Analysis
Copy MoEs from Base	Copies the MOE attributes of key function block into the selected solution block	Architectural Analysis
Perform Trade Analysis	For Weighted Objectives Table, calculates the set of solutions and displays the results in an Excel spreadsheet	Architectural Analysis
Export to New Model	Creates a new model and adds the selected packages and profiles to it	Architectural Design
Import from XML	Exports the existing package structure (with or without diagrams) to an xml file for use as a project template	
Export Project	Creates a new project	
Structure to XML	structure from the selected XML template file (created by the above helper)	

Figure 36: SE Toolkit Features

6 Case Study: Introduction

The Harmony aMBSE process is tool-agnostic; it's perspective is that what tools do is automate or perform activities that the engineer wants to do. This is not to say that tools don't add value. Tools remove tedium from the engineering effort, allowing engineers to focus on those aspects of engineering where they add value. Tools can improve quality by removing sources of human error such as mistakes in transcription or due to lagging vigilance. Good tools are generally process-agnostic, meaning that they provide commonly needed services common to many processes. Of course, it is important that tools and processes be compatible in the sense that they have overlapping needs and services. However, just because a tool automates some aspect of a task doesn't mean that the task is completely done. Nor does it mean that if a tool doesn't automate a task, that the tool is inappropriate for the project. There will always be steps that human engineers perform in every engineering process.

That being said, in this, and the following sections, we will explore a case study using the IBM Rhapsody tool and the Harmony aMBSE Toolkit (aka the "SE Toolkit". Both Rhapsody and the SE Toolkit automate a number of tasks performed by human engineers. This Deskbook will discuss and provide examples of how to use the tooling to achieve your engineering objectives.

The case study in the Deskbook is the **Aircraft Control Surface Enactment System (ACES).** This system receives commands for movement of a rather large set of aircraft surfaces that control the orientation of an aircraft. These moving surfaces are collectively known as "control surfaces" and may be independently rotated – and in some cases, extended and retracted – under command from other aircraft subsystems. See Figure 37.

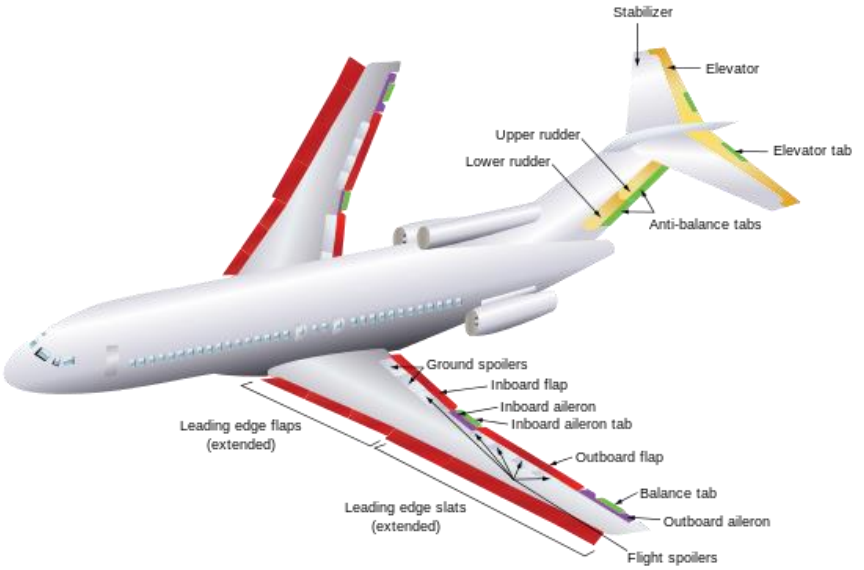


Figure 37: Aircraft Control Surfaces

Some of the control surfaces only rotate. These include: ground spoiler and flight spoiler. Some have a smaller internal and separately controllable surface known as a **trim tab**. These include: inboard wing flap, outboard wing flap, inboard aileron, outboard aileron, upper rudder, lower rudder, and the elevator. Still other control surfaces may also be extended and retracted. These include: the leading edge flaps and leading edge slats. Note that *all* of these control surfaces, with the exception of the rudders, have both left and right side counterparts.

The control surfaces determine the aircraft orientation. The orientation of the aircraft is known as the **attitude** of the aircraft and is defined in three aspects: roll, pitch, and yaw. See Figure 38.

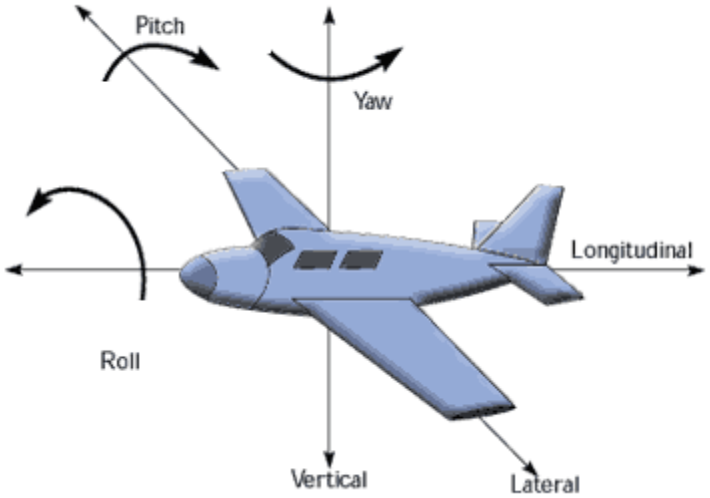


Figure 38: Aspects of Aircraft Attitude

The responsibility for determining what the orientation should be to achieve pilot maneuvering goals are the job of another aircraft system – the **Attitude Management System (AMS)**. The AMS uses an internal set of partial differential equations – known as the **kinematic model** – to compute the set of desired control surface positions necessary to achieve the correct attitude. The fundamental responsibility of the ACES is to move the control surfaces to the commanded positions, maintain them in those positions as forces act on them, and to report on their status.

Other involved aircraft systems include the aircraft electrical power system, the aircraft hydraulic system, and the pilot display. The ACES must receive and distribute electrical power and hydraulic pressure as necessary to execute its duties. The **Pilot Display System (PDS)** will receive some raw data from the ACES, although the bulk of the data display regarding performance of attitude control will come from the AMS so that it can be converted to information directly usable by the pilot.

6.1 Case Study Workflow

Figure 2 shows the overview of the Harmony aMBSE workflow that will be used for the case study. While that workflow includes the additional activities of **Initiate Project and Define Stakeholder Requirements**, those activities will not be employed in this case study. We will begin with system requirements.

Harmony aMSBE Activity	Work Performed	Primary Work Products
System Requirements Definition and Analysis	• Create Requirements • Create use case model • Analyze Control Surfaces use case using system function based worklow[4] • Analyze Start Up use case using scenario based workflow • Create Logical Data/Flow Schema • Create dependability analyses	• System Requirements • Context Diagram • Use case model • Use case execution context • Activity diagram • Sequence diagram • Logical Data Schema • Logical System Interfaces • FMEA • FTA • Security Analysis
Architectural Analysis	• Trade studies	• Parametric Diagrams • Trade study
Architectural Design	• Identify subsystems • Allocate / derive subsystem requirements • Create subsystem use case model • Update logical data / flow	• Subsystem architecture • Subsystem logical interfaces • Logical Data/Flow Schema

[4] See Figure 4 to see the these workflows

Harmony aMSBE Activity	Work Performed	Primary Work Products
	schema • Update dependability analysis • Define cross-subsystem control loops	• FMEA • FTA • Security Analysis
Hand off	• Create Shared Model • Derive Physical Interfaces • Derive physical data / flow schema • Create Subsystem Models • Create deployment architecture for each subsystem • Derive and allocate discpline-specific requirements • Define inter-disciplinary interfaces	• Shared Model • Physical Interfaces • Physical data / flow schema • Subsystem deployment architecture • Software / electronic / mechanical requirements • Inter-disciplinary interfaces

Figure 39: Case Study Workflow

We will focus on two use cases in this case study. The first use case, **Start Up**, will use the System Function-Based use case analysis workflow. The second, **Control Air Surfaces** will use the Scenario-Based use case analysis workflow in Figure 4.

Figure 40 shows the overall case study workflow.

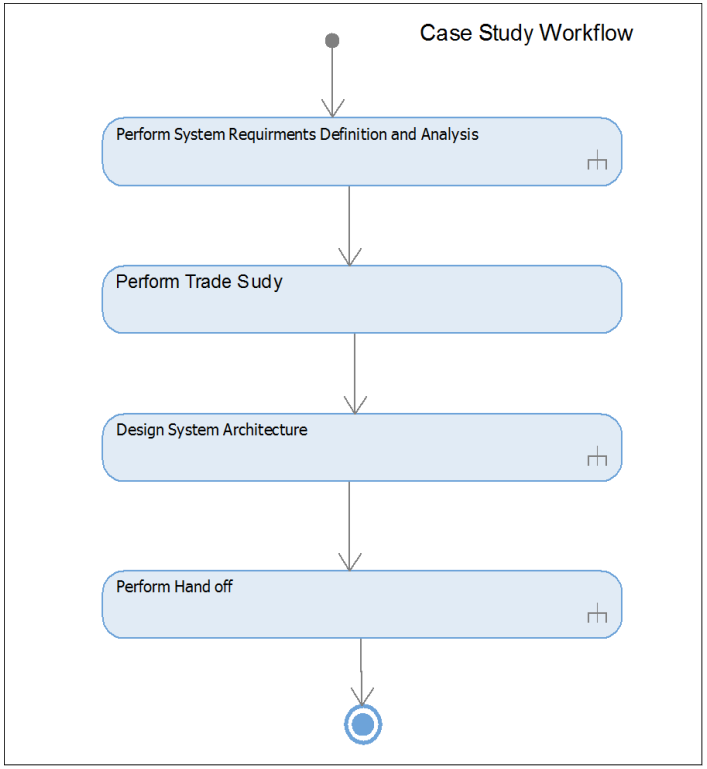


Figure 40: Overall Case Study Workflow

Figure 41 shows the details of the primary activities to be done in the definition of requirements and the analysis of use cases for the case study.

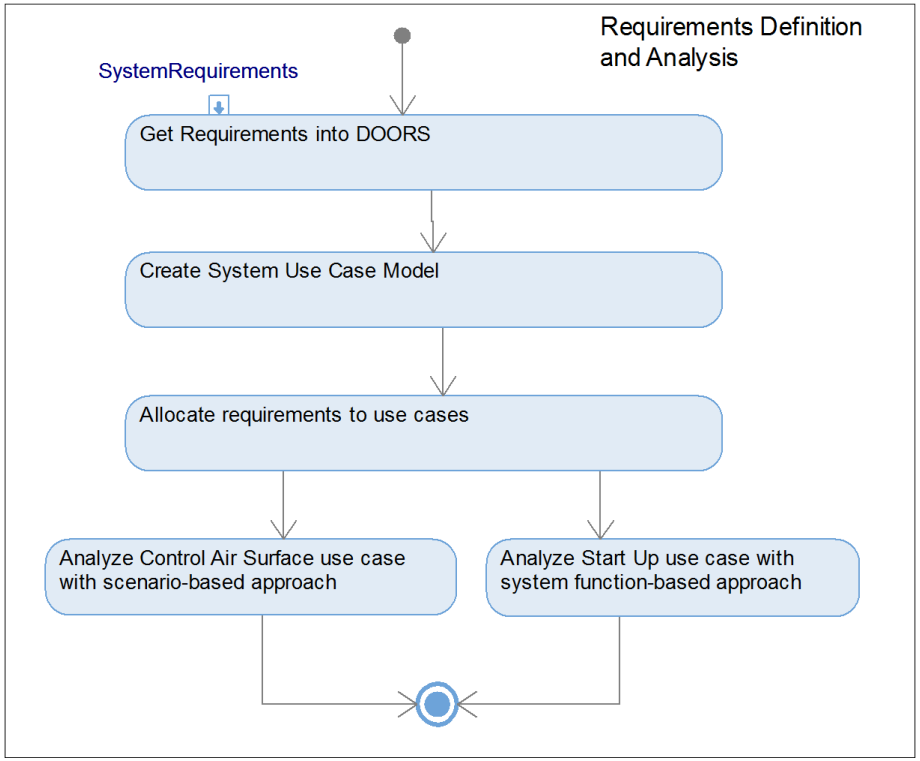


Figure 41: Case Study Requirements Definition and Analysis Workflow

Figure 42 shows the detailed actions to be performed during the architectural design of the case study. In the case study, two different approaches will be taken to allocating the requirements for the two system use cases under consideration.

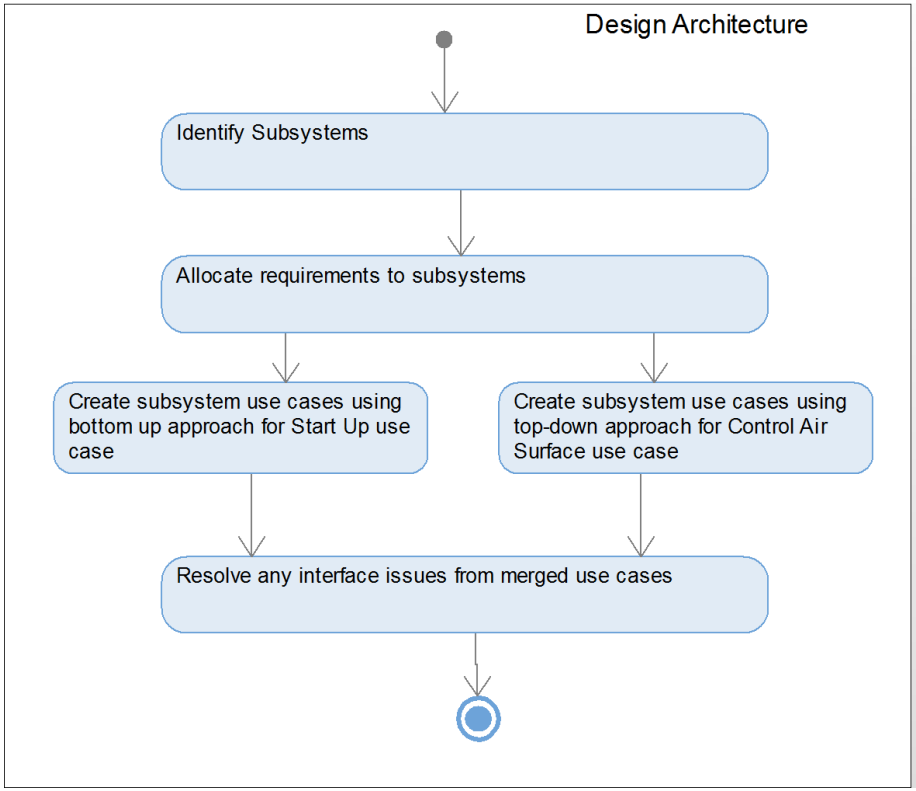


Figure 42: Case Study Architectural Design Workflow

Lastly, Figure 43 shows the hand off workflow for the case study. In the case study, we will create a shared model that refines the logical interfaces from the two analyzed use cases and creates physical interfaces and data schema from the logical specifications. Then a single subsystem model will be created (of the several that would be created in a real project). This subsystem model will then be detailed by creating a deployment architecture for the subsystem and requirements will be allocated to those disciplines. Finally, the interfaces between the engineering disciplines will be defined.

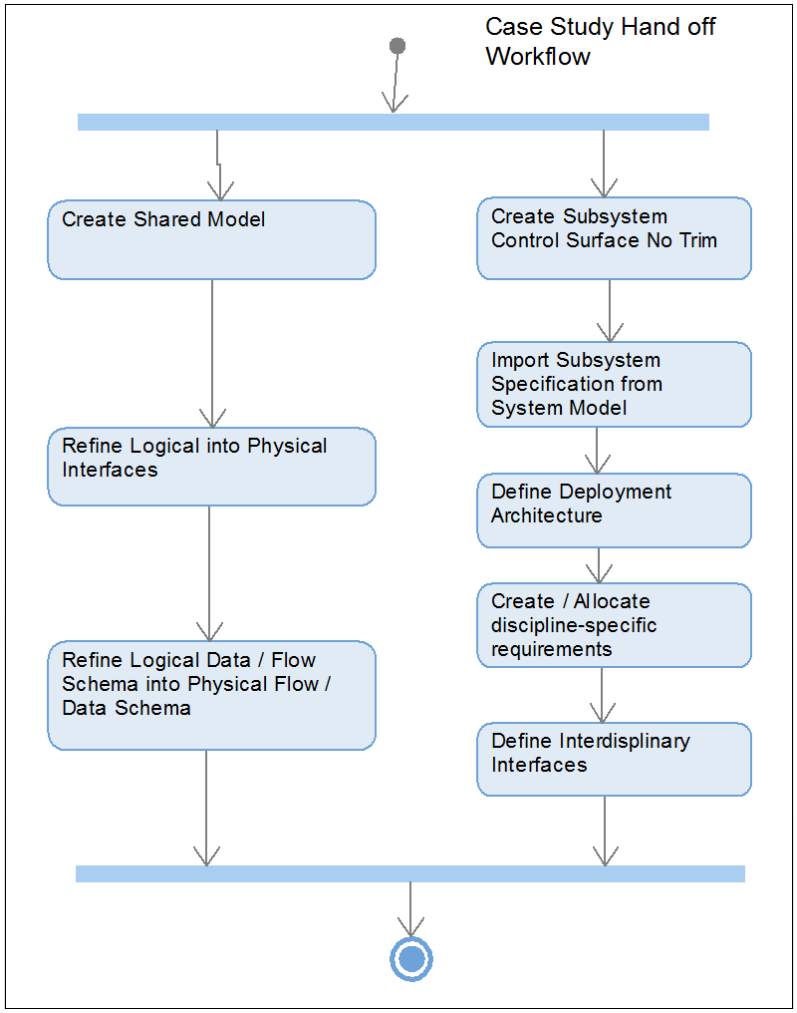


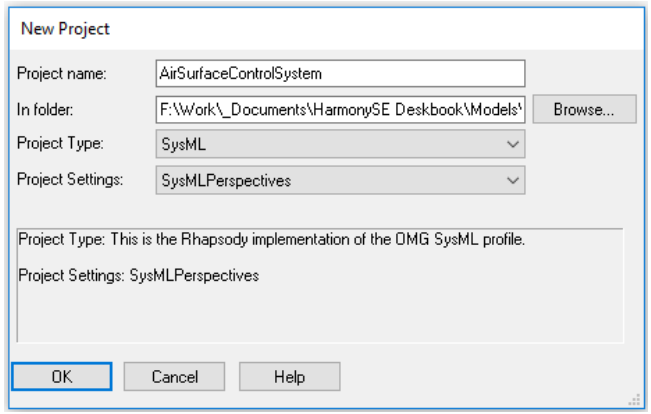
Figure 43: Case Study Hand Off Workflow

At this point, the systems engineering work for the case study is complete and has resulted in specifications that downstream engineering teams can take and begin the detailed design and implementation of the subsystems.

6.2 Creating the Harmony Project Structure

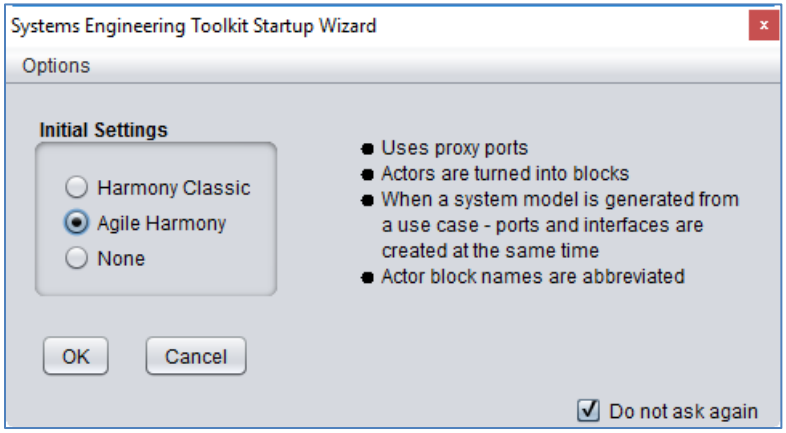
The Harmony aMBSE process recommends a particular project structure that has proven to be useful. Once an initial Rhapsody model has been created, this can be quickly done with the SE Toolkit feature **Create Harmony Project.**

- Start Rhapsody
- In the main menu select *File > New* and enter the project name (e.g. **AirSurfaceControlSystem**) and click on the *Browse* button to select the directory for its placement.
- Under the *Project Type*, select **SysML**. Under *Project Settings*, select *SysML Perspectives*.

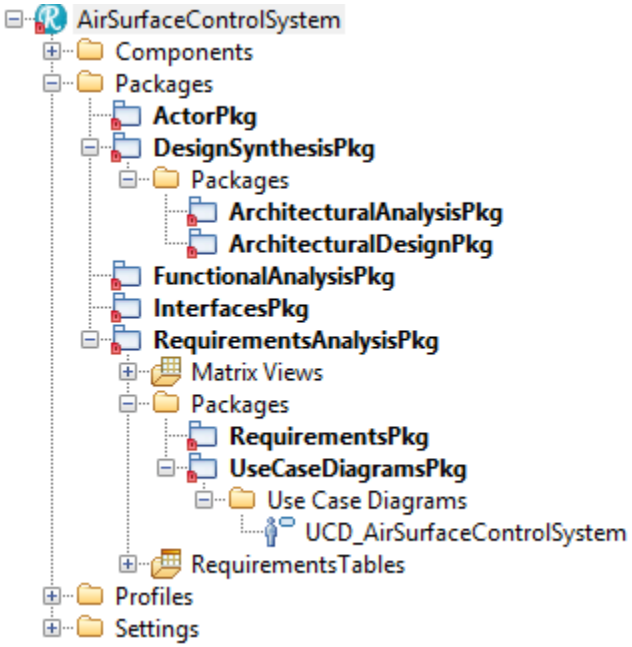


- Click on the OK button. If a diagram appears asking if you want to add the SysML Perspectives, click on *Yes*.
- If a dialog appears asking if you want to create the project directory, click on *Yes*.
- Select *File > Add Profile to Model* and double-click the **HarmonySE** directory, then double-click again on the HarmonySE.sbs file.

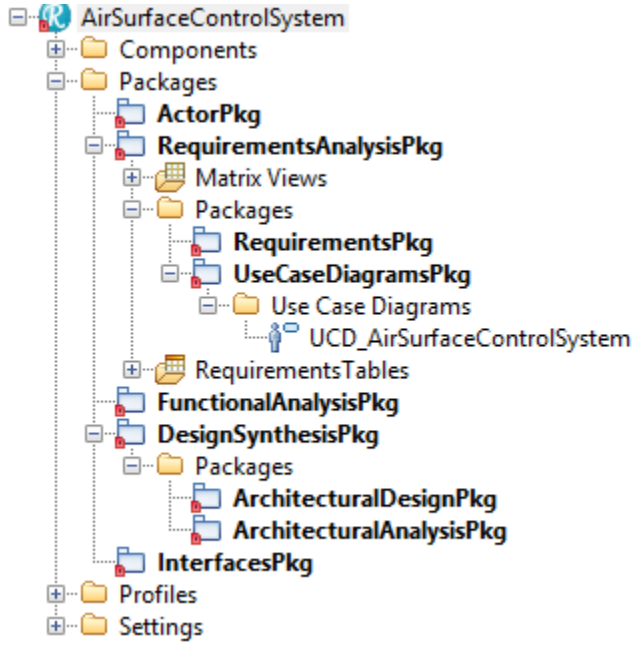
- A dialog will appear for initial settings. Select **Agile Harmony**, select the *Do not ask again* checkbox and click on the *OK* button.



- Now right-click on your project name in the browser and select *SE Toolkit > Create Harmony Project*. The project browser will now have the following structure:



- If desired, you can reorder the packages in the browser by selecting *View > Browser Display Options > Enable ordering*. Once selected, you can then select a package and using the up and down arrows of the browser to order the packages as you like. This is the ordering that I prefer:



We are now ready to begin the engineering work on the case study.

7 Case Study: System Requirements Definition and Analysis

The objectives of this phase of the Harmony aMBSE process are to

- Get requirements into the Rhapsody model
- Create the overall use case model
- For each use case
    a. Allocate relevant system requirements to the use case
    b. Identify and correct requirements that are missing, incorrect, inconsistent or inaccurate by constructing a high-fidelity model of the use case
    c. Define the logical interfaces between the system in the context of the current use case and the actors
    d. Create a data and flow schema for data and flows used in the logical interface
    e. Perform dependability analyses to identify relevant safety, reliability, and security concerns and requirements.
- As necessary, resolve interface inconsistencies between the use cases

We will follow the overall workflow capture in Figure 3.

7.1 Get System Requirements Into Rhapsody

For the purpose of this case study, we will import the requirements into our model from the Rhapsody project **ACES_ReqsOnly**. This model has a package named **SysReqsPkg** with some subpackages containing the set of system requirements.

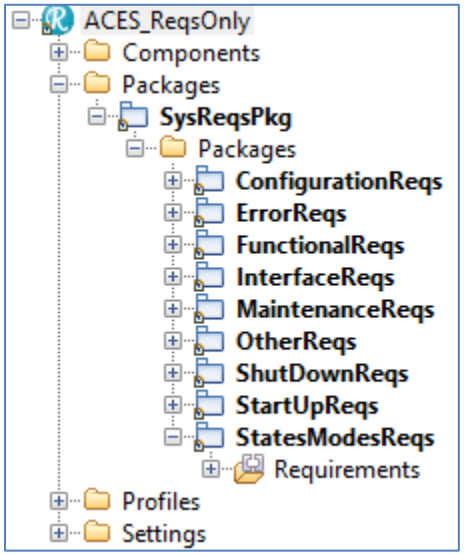


Figure 44: Packages in ACES_ReqsOnly model

To add the requirements, go to *File > Add to Model*. Then navigate to the location of that model in your hard disk. Then go to the **ACES_ReqsOnly_rpy** subdirectory and select the file **SysReqsPkg.sbs** in the dialog. Be sure that *Add Subunits i*s checked and you've selected *As Unit* (**not** *As Reference*). Click on *Ok* to add the package to your model. This will add the package and the nested packages and requirements.

Note that in real projects, it is far more common to import the requirements from a requirements management tool such as DOORS or DOORS NG. However, in this Deskbook, we are focusing on the modeling aspects.

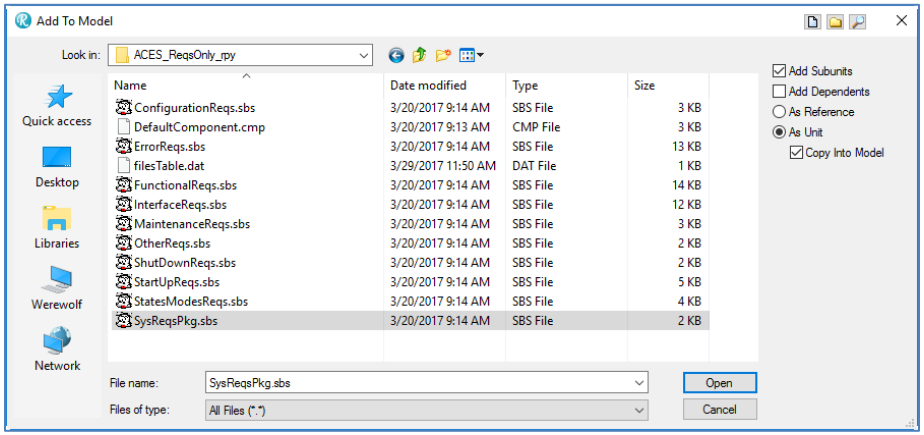


Figure 45: Adding the System Requirements package to your model

- Now select all the nested packages under **SysReqsPkg** and drag them to their expected location **in RequirementsAnalysisPkg > RequirementsPkg**.

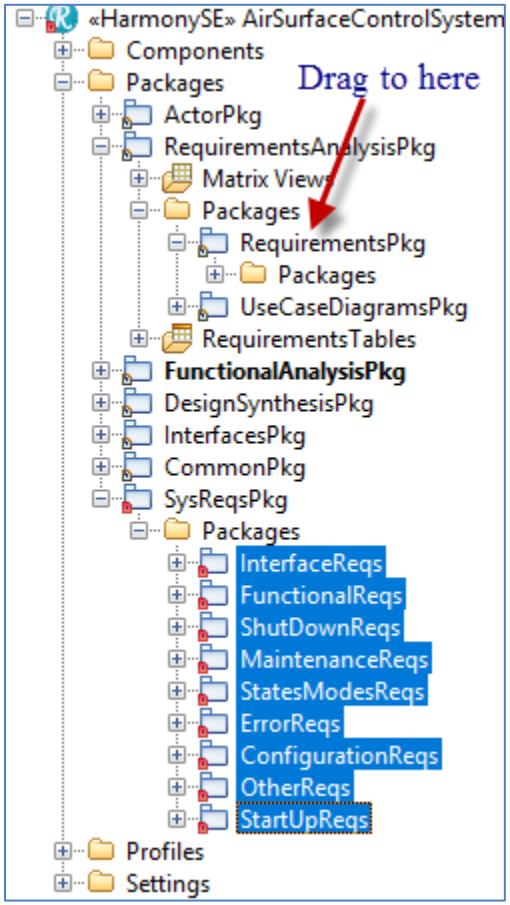


Figure 46: Preparing to drag requirements packages to RequirementsPkg.

- You may delete the now empty **SysReqsPkg**.

7.2 Create the System Use Cases

This activity corresponds to the *Identify System Use Cases* task in Figure 3.

When you used the *Create Harmony Project* tool, the SE-Toolkit created an empty use case diagram. Unless you've closed it, it should be open in a tabbed window. If it is not currently open, navigate in the browser to

RequirementsAnalysisPkg > UseCaseDiagramsPkg > UseCaseDiagrams > UCD_AirSurfaceControlSysem. Double click on the diagram in the browser to open.

In this case, the following use cases have been identified from the system requirements:

- Start Up
- Shut Down
- Control Air Surfaces
- Manage Power
- Configure System
- Manage Data
- Update Status

Each of these is an important and complex system usage with several to many requirements and interesting scenarios.

Using the tools in the use case diagram tool bar, create the use case diagram shown in Figure 47.

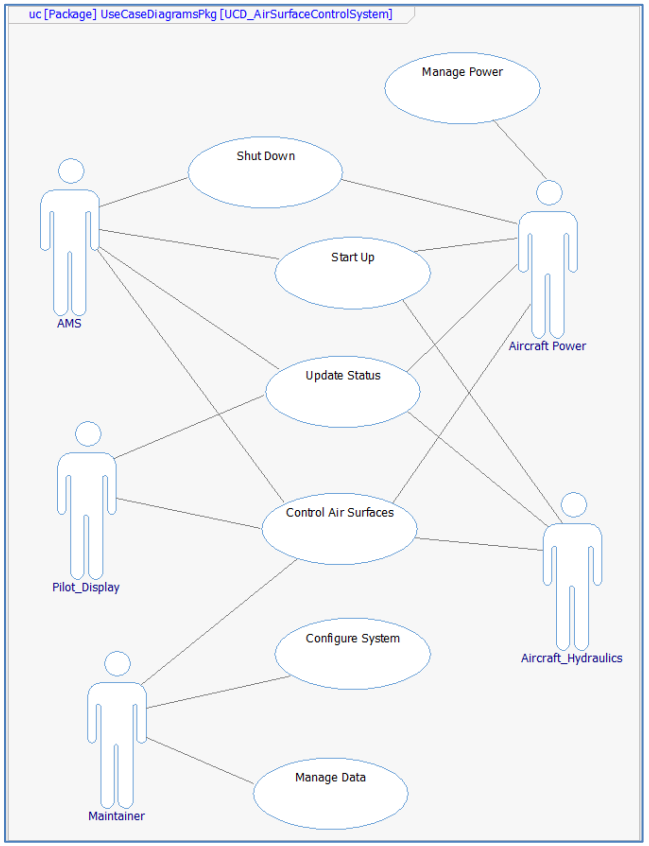


Figure 47: System Use Cases

A (very) short description of the objectives of the use cases:

- **Start Up**: Manages the start up process, including cold and warm states, and, in the case of cold start, the Power On Self Test (POST).
- **Shut Down**: Manages an orderly shut down of the system, including zeroing the positions on all surfaces.
- **Manage Power**: Manages the electrical power delivered to the system from the aircraft, including the selection of the power source.
- **Update Status**: Periodically updates the AMS and Pilot Display as to the operational state of the system, including statuses for all the

control surfaces, the hydraulics, the electrical power, and operational flight mode.

- **Control Air Surfaces**: manages the response to AMS commands for control surface position changes, performs station keeping at the commanded position, and identifies positional accuracy and timing errors.
- **Configure System**: sets range limits for the control surfaces, range and time accuracy limits, and allows for software upgrades.
- **Manage Data**: supports storage and download of stored operational data, including fault and failure information.

In this case study, we will limit our discussion to the **Control Air Surfaces** and **Start Up** uses cases only. Interested readers should feel free to model the other use cases at their leisure.

7.2.1 Add use case mini-specification

Note: to use this description wizard, you will need to apply the *«HarmonySE»* to the Rhapsody project (double click on the project name in the browser and select the stereotype in the stereotype drop down list).

Let's add a mini-specification to these two use cases. The SE Toolkit provides a tool for this. Right-click the **Start Up** use case and select *SE-Toolkit > Import Description from RTF*. This provides a standard template which you can elaborate for the description fields of various kinds of model elements. The default template includes places for a short explanation of its purpose, description, security constraints, preconditions, post-conditions, and invariants (assumptions).

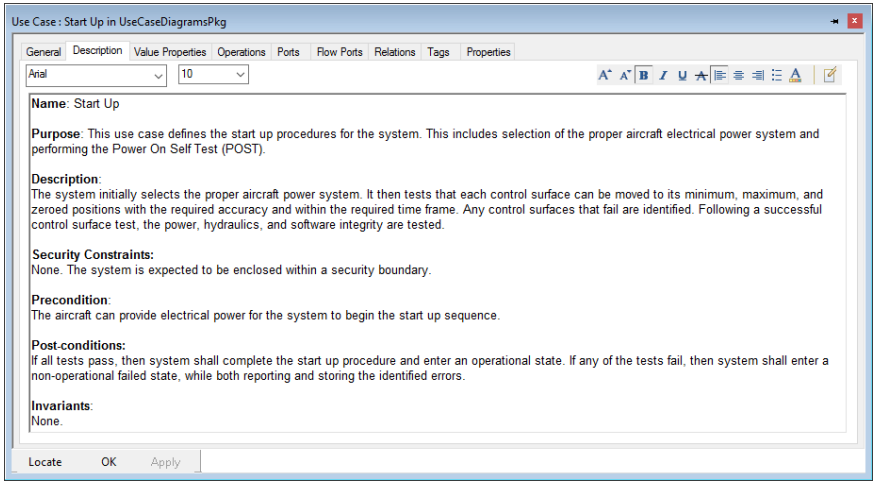


Figure 48: Start Up Use Case Description

Add a similar description of the **Control Air Surfaces** use case.

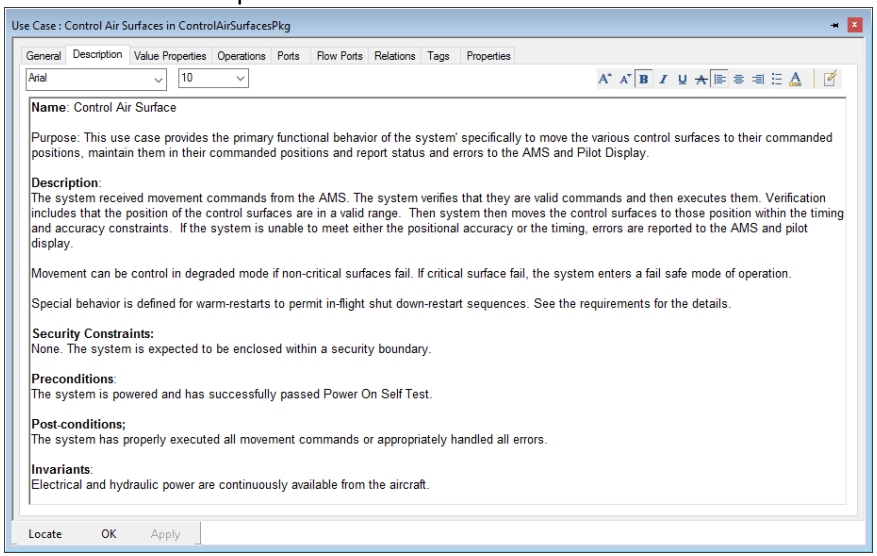


Figure 49: Control Air Surfaces Use Case Description

7.2.2 Allocate requirements to the use cases

Each use case must be linked with the functional and quality of service requirements it collectively represents. This can be done in multiple ways.

Let's do one diagrammatically and one using the pre-defined **Use Case Trace Matrix Layout** in that is predefined in the *HarmonySE* profile.

7.2.2.1 Adding Traces on a Use Case Diagram

Creating dependencies on a diagram is easy and it provides a nice visual reference of the traced requirements.

- In the **RequirementsAnalysisPkg > UseCaseDiagramsPkg**, add a new use case diagram.
- Name this diagram **Start Up Use Case Requirements**.
- Drag the use case Start Up on to it.
- Now, drag the appropriate requirements from the **RequirementsAnalysisPkg > RequirementsPkg** on to the diagram (All the requirements in the **StartUpReqs** package plus the **StateModeReq_1** from the **StatesModesReqs** package - see Figure 50).

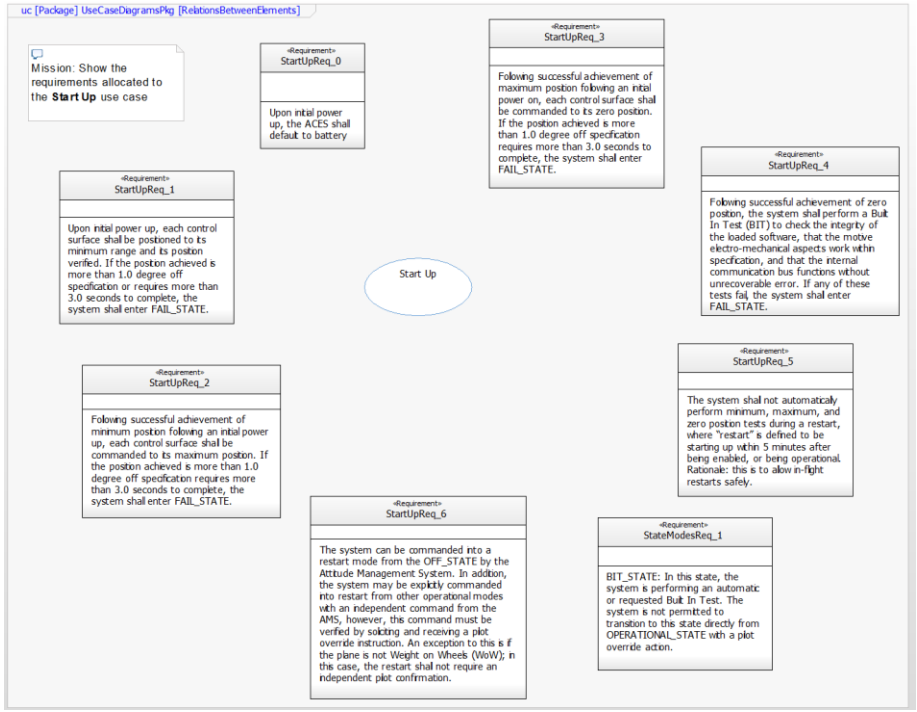
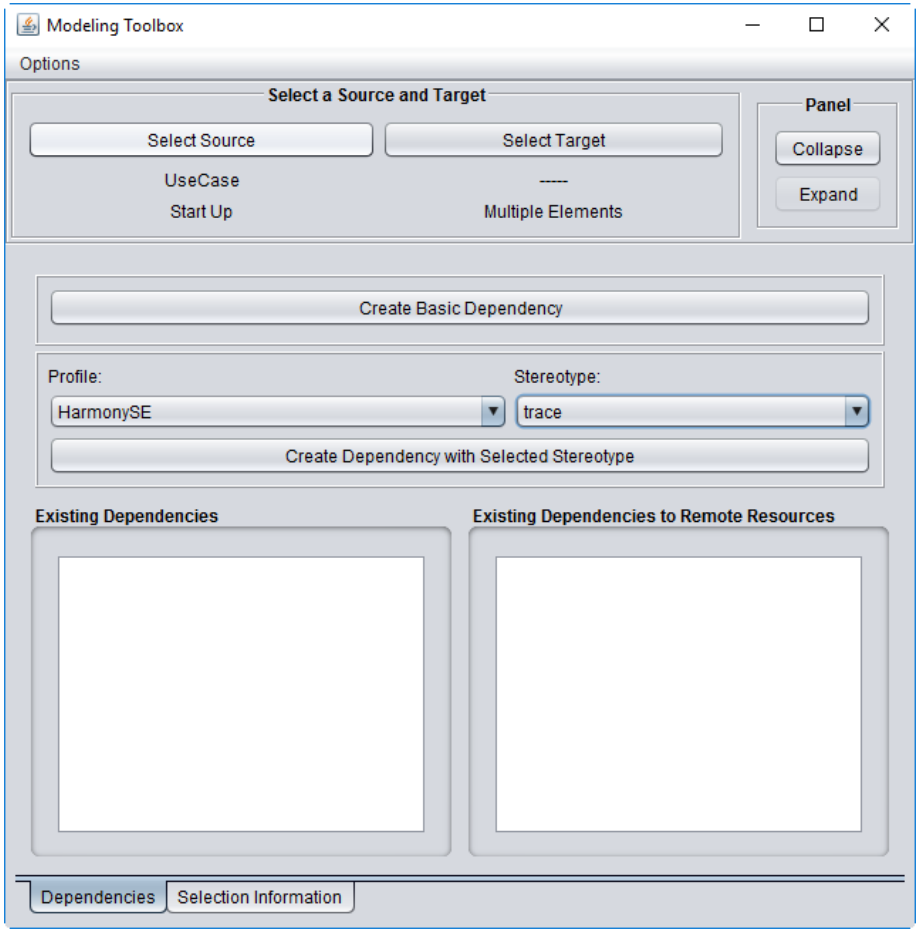


Figure 50: Adding Use Case Trace relations diagrammatically – step 1

- Right click on the use case in the diagram and select *SE Toolkit > Add Dependencies > From Selected*. This will open the Modeling Toolbox dialog. (NOTE: Yes, the trace relation goes FROM the use case TO the requirement!)
- Now select all the requirements (select the first requirement, then click on the others one at a time with the control key depressed).
- Once all the requirements are selected, click on the *Select Target* button in the *Modeling Toolbox* dialog.
- Next, select the *HarmonySE* profile from the *Profile* drop down list on the diagram and the *trace* stereotype in the *Stereotype* drop down list on the dialog.

The *Modeling Toolbox* dialog should now look like this:



- Click on the *Create Dependency with Selected Stereotype* button.

The relations may not show in the diagram. If not, while the diagram has focus using the menu *Layout > Complete Relations > All* to show the elaborated diagram. It should look like this:

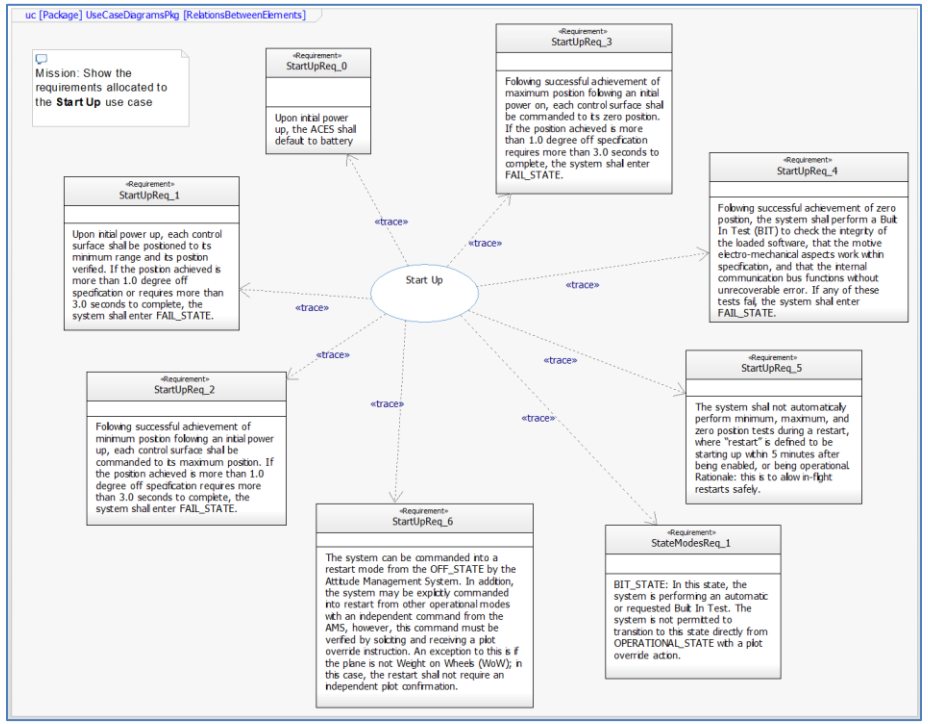


Figure 51: Adding Use Case Trace relations diagrammatically – complete

7.2.2.2 Adding Traces using the Use Case Trace Matrix

The other approach is to do this in a matrix. The Harmony SE toolkit provides such a matrix layout. In fact, the toolkit adds a layout for you in the **RequirementsAnalysisPkg** when you used the *SE Toolkit > Create Harmony Project* tool previously.

- Double-Click on the matrix view to open it up.
- Because there are many more requirements than use cases, click on the *Switch Rows and Columns* tool option (normally located to the right of the open view).

If you scroll through the matrix, you will see the trace relations we added in the previous step for the **Start Up** use case.

Add the traces

- Trace all the functional requirements (with identifies **FuncReq_0** through **FuncReq_40**) to the **Control Air Surfaces** use case. To do this,
  - In the row in the matrix labelled **Control Air Surfaces** (assuming you previously toggled rows and columns), Select all the corresponding cells
  - Right click and select *Add New > trace*.
- Similarly add **ErrorReq_0** through **ErrorReq_36** in the same fashion.

You've now successfully traced from the **Control Air Surfaces** to the relevant 76 requirements. A portion of the matrix is shown in Figure 52.

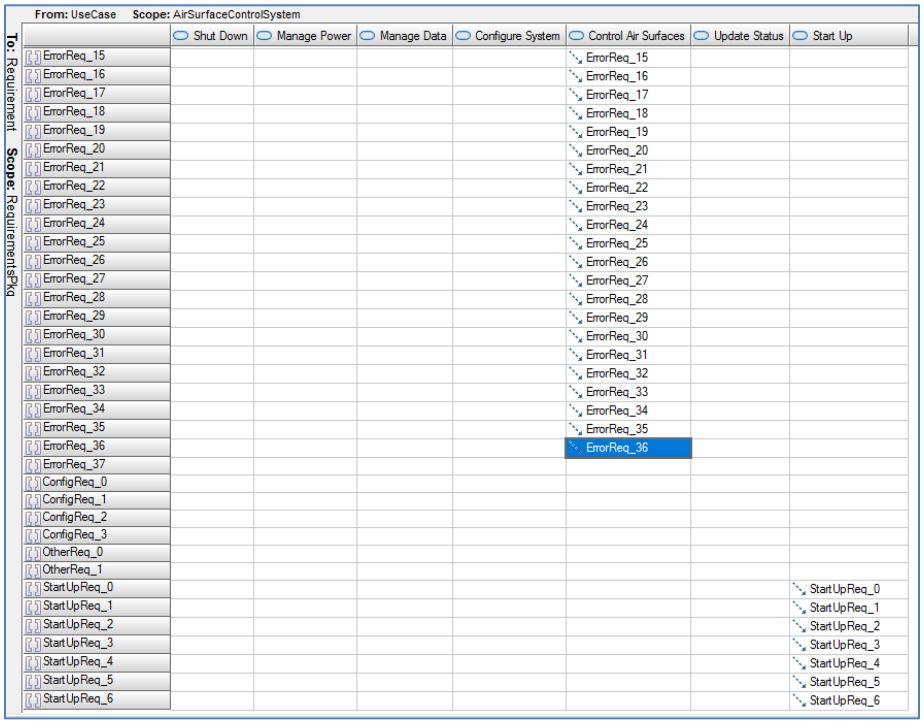


Figure 52: Portion of the Use Case - Requirements trace matrix

Let's now analyze the two use cases to identify missing, incorrect, inaccurate, or inconsistent requirements.

7.3 Analyze the Start Up Use Case

We're going to analyze two use cases. The first, and simpler of the two, is the **Start Up** use case. This use case is concerned with how the system goes from off to ready to operate. Most of the behavior for this use case is focused around the executing the Power On Self Test (POST) and managing its outcomes. We will analyze this use case using the *System Function Based Approach* from Figure 4. We'll create an activity diagram to organize the various actions (system functions) associated with the use case. From that we'll use the Harmony SE Toolkit to generate the scenarios. Then we'll construct an executable state machine that simulates the system functions and the system interaction with the system actors as a means to verify the quality and completeness of the requirements.

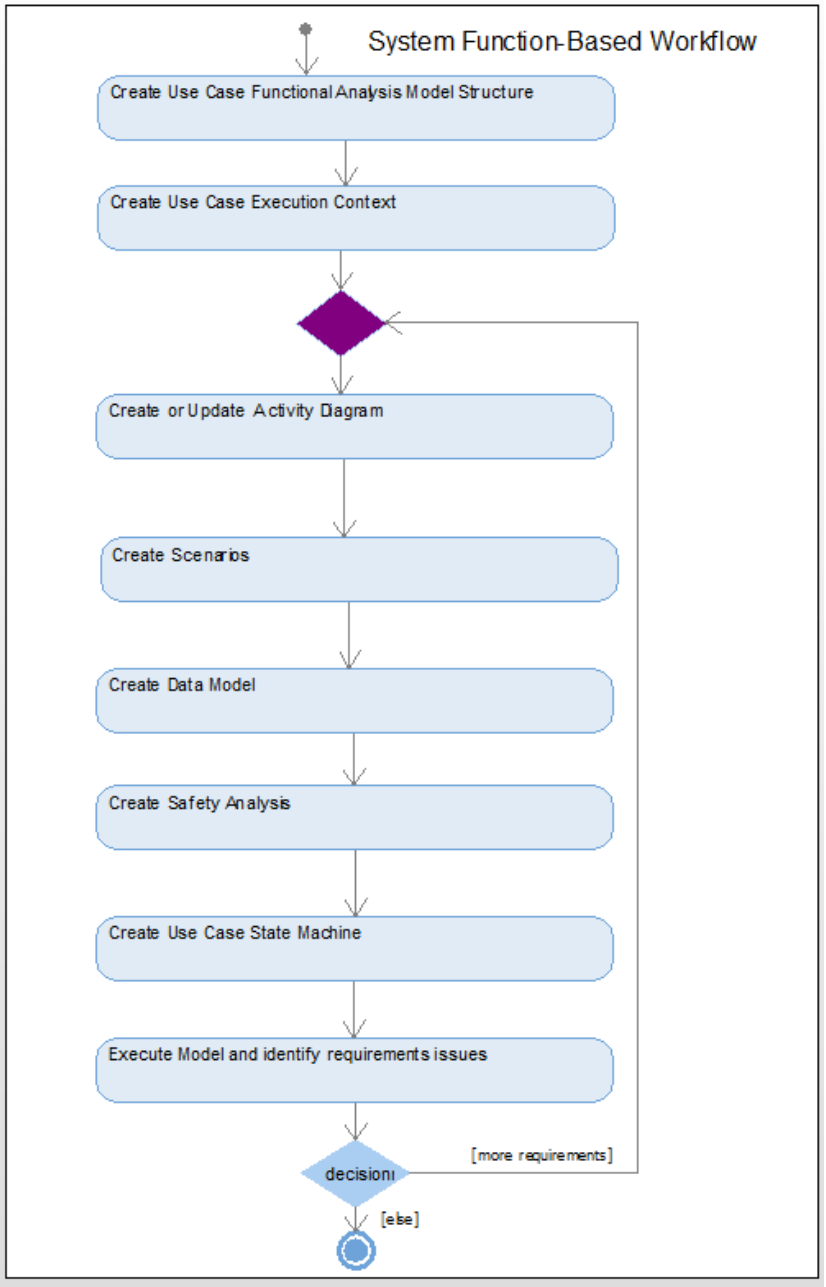


Figure 53: Detailed Workflow for System Function-Based Analysis

Let's get started.

7.3.1 Create Use Case Functional Analysis Model Structure

First, we'll set up the model structure using the SE Toolkit. On the use case diagram or the browser, right-click the Start Up use case and select *SE-Toolkit > Create System Model From Use Case* (Figure 54).

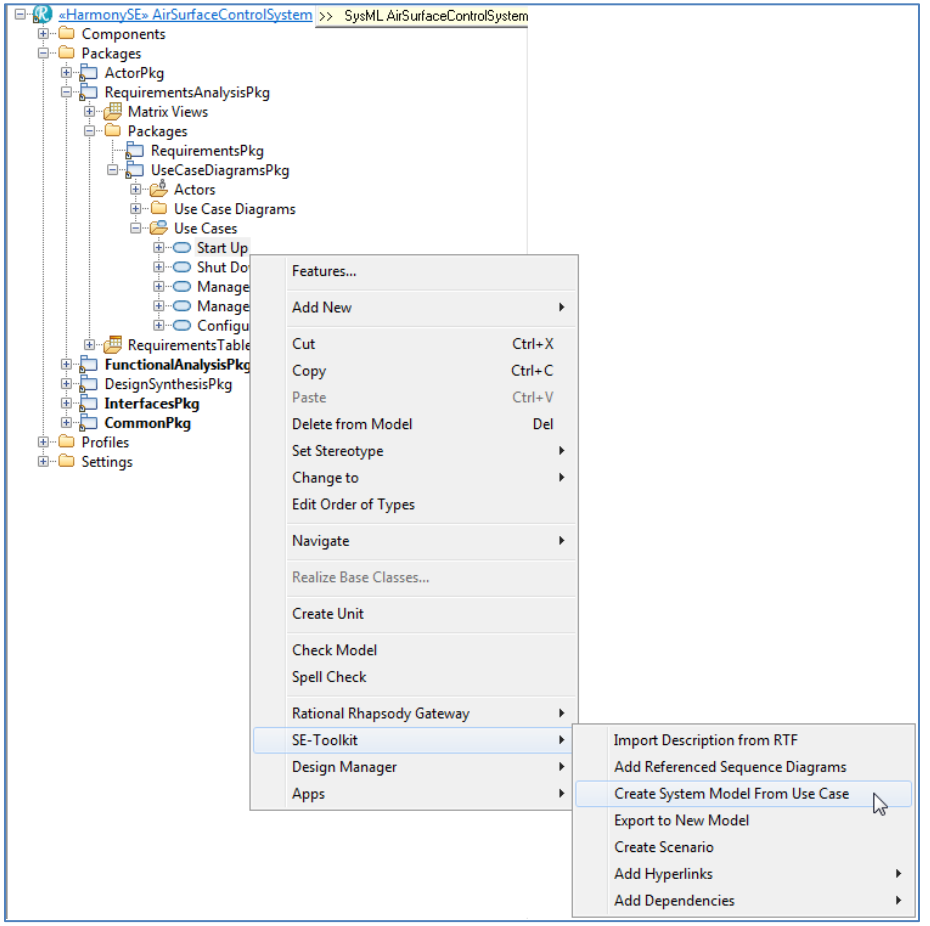


Figure 54: Create System Model from Use Case

This tool creates a package called **FunctionalAnalysisPkg > StartUpPkg** and then populates it with the appropriate blocks for the use case and actors, creates the appropriate links and even creates a new internal block diagram (IBD) showing the use case execution context. The **StartUpExecutionScopePkg** also contains a new component named **StartUp_Sim** for building the executable model (to come later). The fully elaborated package structure for this functional analysis package is shown in Figure 55.

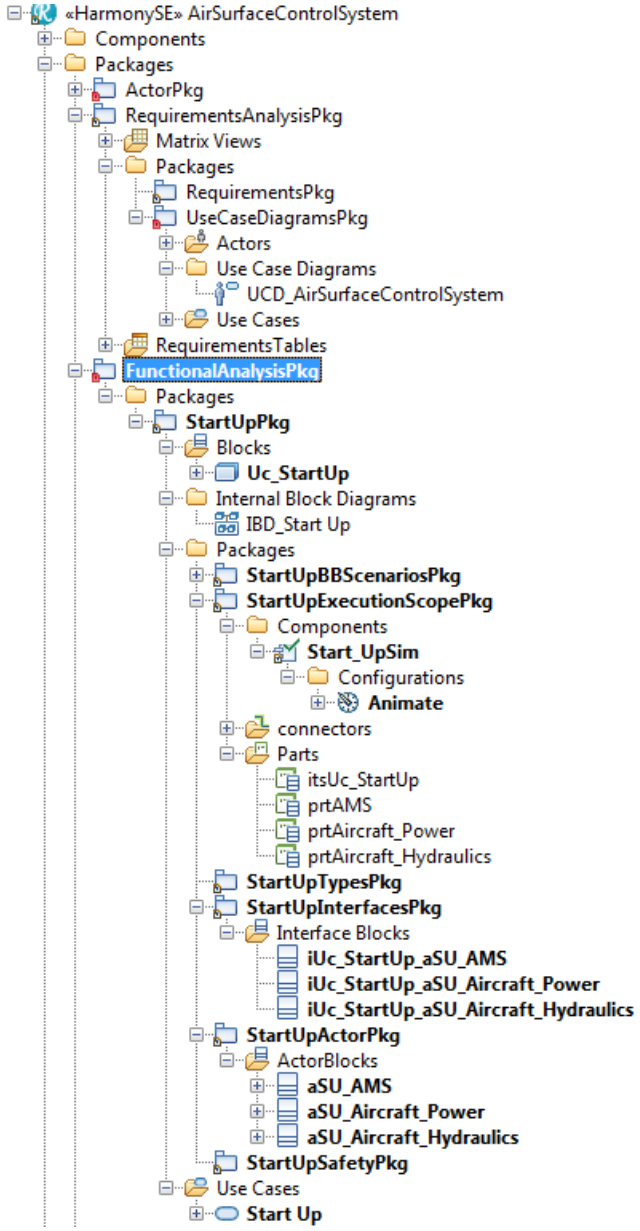


Figure 55: Start Up Use Case Analysis Model Structure

There are a couple of interesting things to note in this structure. First, the "actor blocks" (i.e. blocks derived from the use case actors) are named with '**aSU_**' prepended to the original actor name. These actor blocks represent custom versions of the actor to support construction and execution of specific use case simulations without affecting any other use case, even if that other use case also uses the same system actor.

Secondly, while the tool creates a default IBD, it isn't very pretty. This is due to limitations in the Rhapsody tool API. You must open the diagram and manually resize and reorient the elements to beautify the diagram. The IBD resulting from this beautification effort is shown in Figure 118:

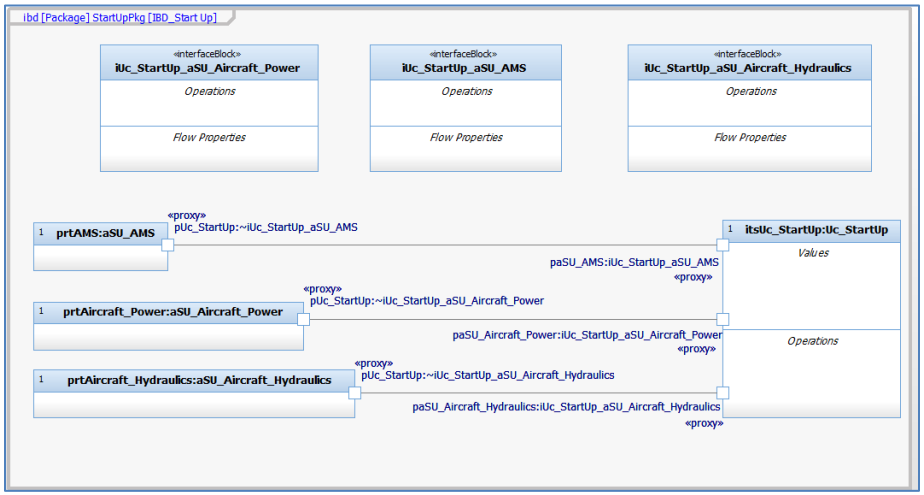


Figure 56: Beautified Use Case Execution Context IBD

The default interface block names may see a little long; you should feel free to shorten them as you like.

7.3.2 Create the Activity Diagram

The requirements spell out what is required for the system start up:

- If the elapsed time since the last start was less than 5 minutes, go directly to WARM state, ready to go directly to operational mode when commanded, otherwise:
  - Switch to battery power from whatever power source is currently being used
  - Move each control surface to its minimum and maximum positions, verifying the accuracy and timing of the movements
  - Zero each control surface position, verifying the movement accuracy and timing
  - Verify the power is within specified limits
  - Verify the hydraulic pressure being provided by the aircraft hydraulic system is within limits and there are no internal pressure losses
  - Verify the integrity of the software
  - If all the tests pass, then proceed to the WARM state; otherwise do not.

The SE Toolkit has created an empty activity diagram for you to elaborate the activity view. You can open it by navigating to **FunctionalAnalysisPkg > StartUpPkg > Use Cases > Start Up > StartUpBlackBoxView > Activities** and double clicking on **activity_0**. Here you can add activities, decisions and flows from the diagram toolbar.

A little bit about naming conventions

The two most common naming conventions for compound names are to use upper case words separated with underscores and to use what is called "camel case." An example of the former approach is **Determine_Time_Since_Last_Restart**. The latter is the practice of writing names by removing the white space between the words but making each new word upper case, as in **DetermineTimeSinceLastRestart**.

Complicating the naming rule is the common practice of beginning the names of types (such as blocks and use cases) with upper case (such as **ErrorReport**) but the names of features of types (value properties/attributes and operations) and instances with lower case (such as **myErrorReport** or **ErrorReport.errorNumber**).

Whichever you choose is fine, but you should be consistent.

Indicating input and output events to/from Actors

There are two ways to show inputs and output events on activity diagrams. The standard UML/SysML way is to use *Send Action* and *Receive Event Action* from the toolbar. This works fine but the latter does not identify the source of the event. The Harmony Profile adds the notion of an *Actor Pin* for an action.

To do this, add a normal action, right click and select *SE-Toolkit > Add Actor Pin*. This will bring up a dialog where you can specify the actor with a drop down list and the direction (*in*, *out* or *both*). The actor pins are used in the automatic generation of sequence diagrams from activity diagrams, which will be used later.

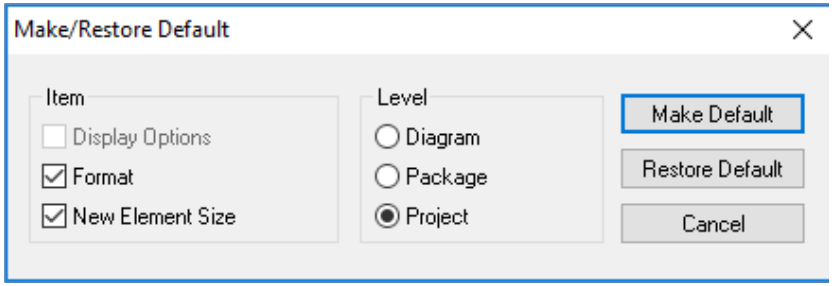
The high level activity diagram for this behavior is shown in Figure 57. Because running the tests involves a large number of actions, the **Range_Surface_Test** and **Perform_BIT** actions on Figure 57 are *call behavior actions*, and there details are shown on other diagrams (Figure 58 and Figure 59).

To add the call behavior actions on Figure 57, simply

- Add a regular action
- Name the action **Range_Surface_Test**
- Right-click on the action and select *SE-Toolkit > Create Call Behavior*.
- Delete the original action you added.

Repeat this process to add a call behavior for **Perform_BIT**. Subsequently, clicking on the fork icon in the action box will directly open the activity diagram it now references. Now you can elaborate the behavior on those referenced activity diagrams

See the different colors for the *Decision* and *Merge* nodes on the activity diagams? This isn't the default, but you can make it the so, by adding a *Merge* node to the diagram, coloring and sizing it as you like, then right-click on it and select *Make/Restore Default…* This will open a dialog that allows you to make this the default format and size for the element within the selected scope.



I set my scope to Project.

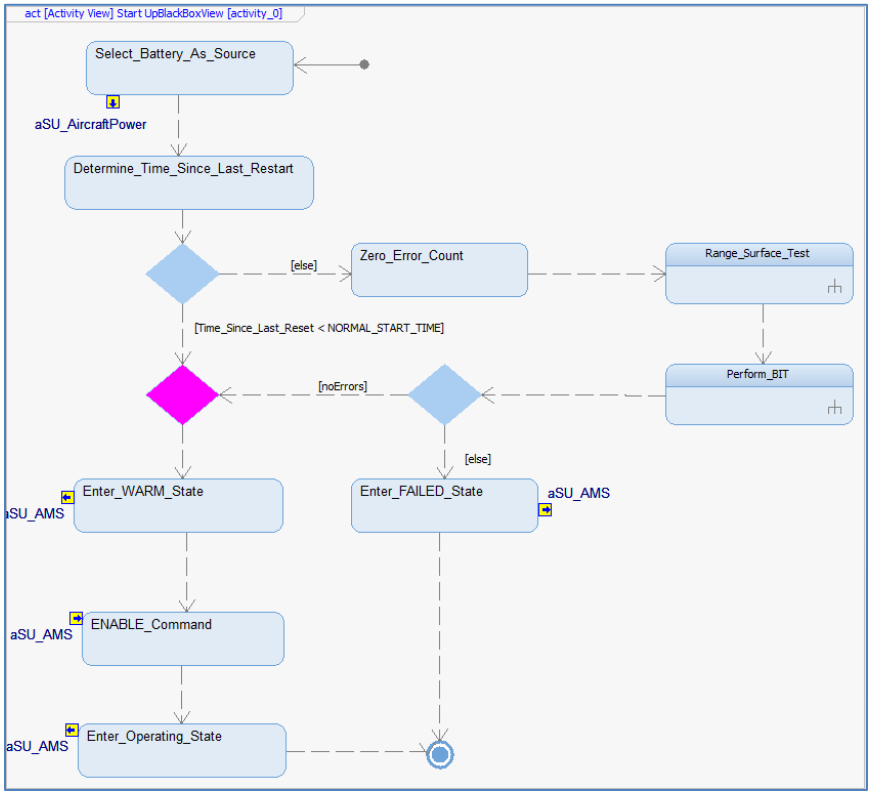


Figure 57: Start Up Use Case High Level Activity Diagram

.

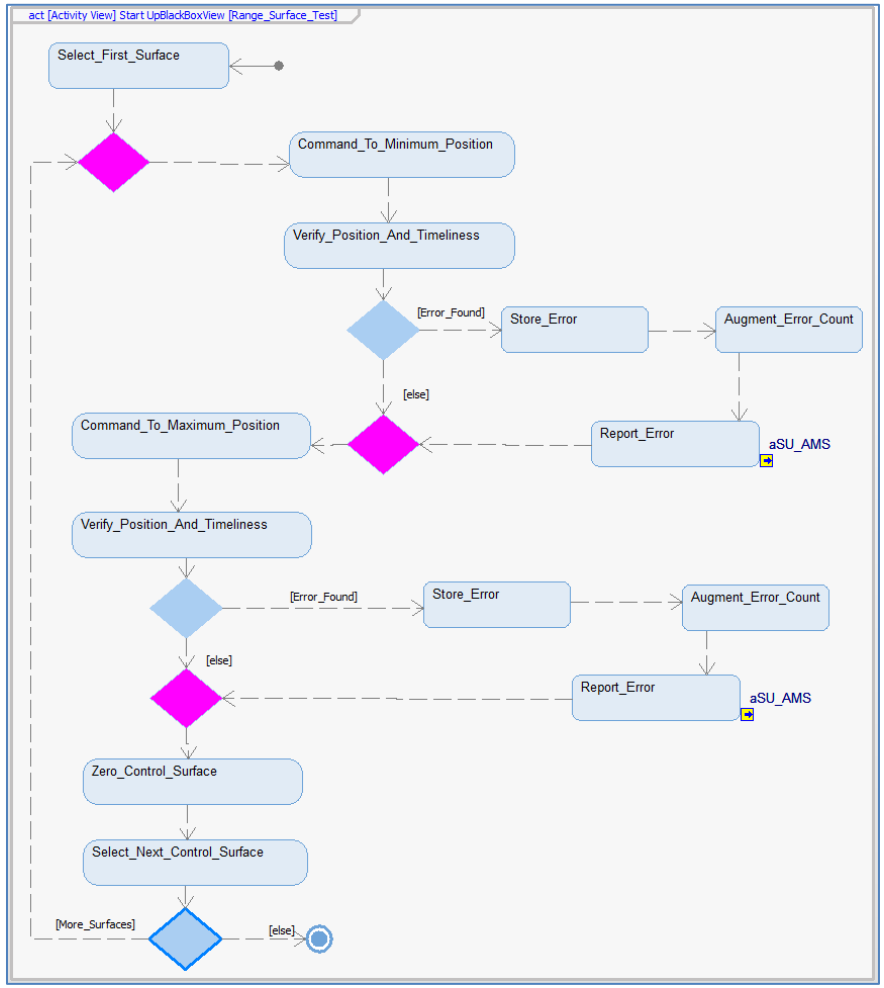


Figure 58: Range Surface Test Activity

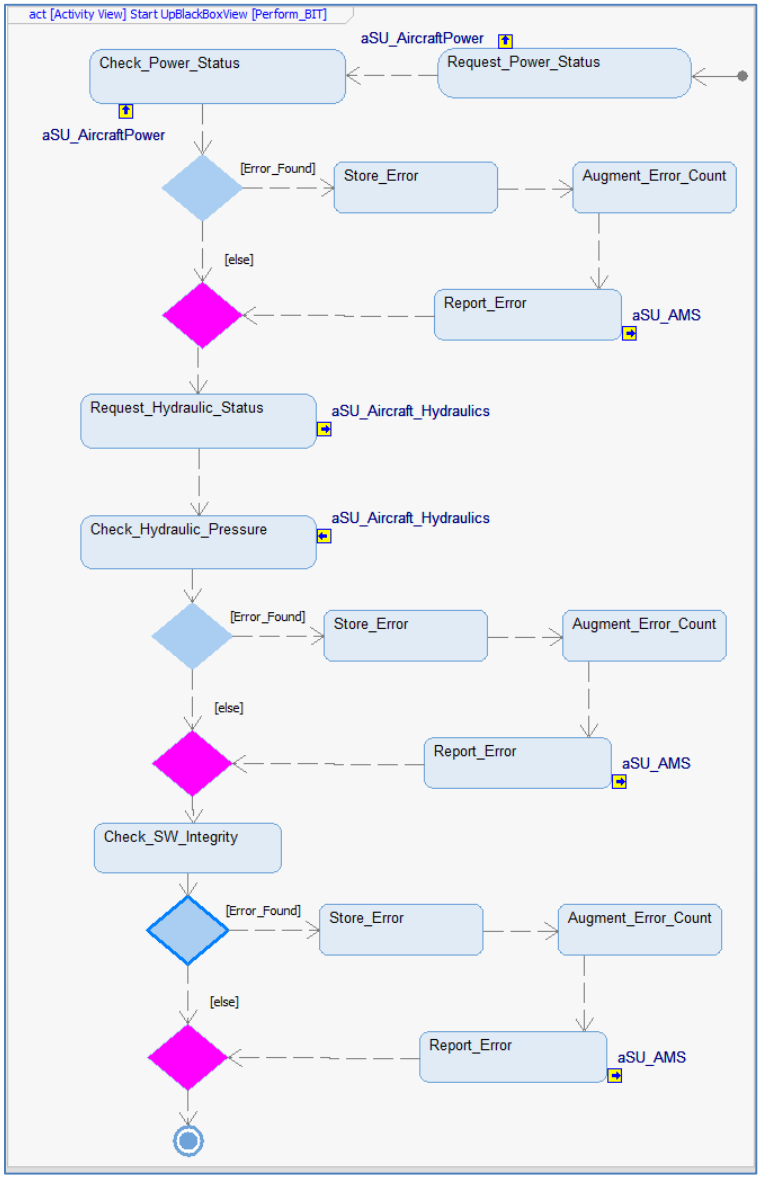


Figure 59: Perform BIT Activity

7.3.3 Generate Scenarios from the Activity Diagram

The next step in this workflow is to generate scenarios. Fortunately, the SE Toolkit has a tool that saves lots of time and effort. To use it, simply right-click on the activity diagram and select *SE-Toolkit > Generate Sequence Diagrams*.

The diagrams created in this way follow a single flow, so you will have to provide guidance as to which path when multiple paths are available, such as at decision points.

Note: although it is possible to run the **Range Surface Tests** for all the surfaces, in practice it is enough to do a single one, but be sure to generate both successful and unsuccessful test cases at all test case branch points.

When you select the tool, a modeling toolbox dialog pops up to allow you to guide the process (Figure 60). In this first example, we'll create a sequence diagram that shows the flow for a warm restart.

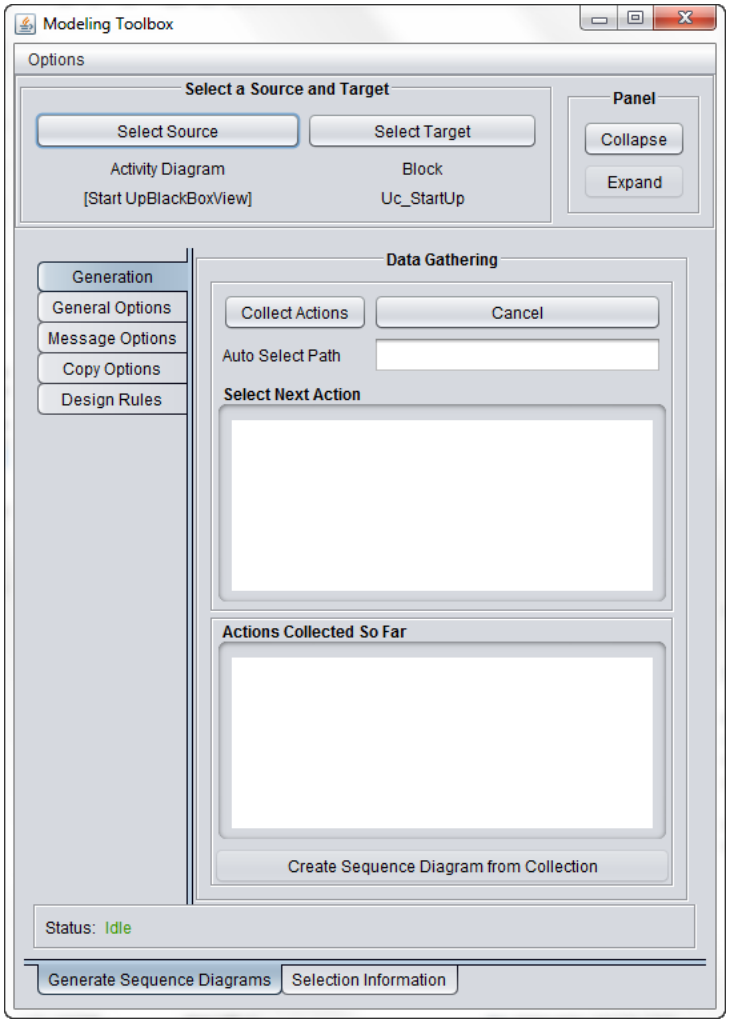


Figure 60: Dialog for Generating Sequence Diagrams

For the tool to proceed, you must see if you need to apply design rules. Because Rhapsody builds executable models, it is picky about naming. The design rules allow the tool to force the names of the actions to conform to the rules. If you select the *Design Rules* tab in the dialog, you'll see that it has identified some invalid naming of actions:

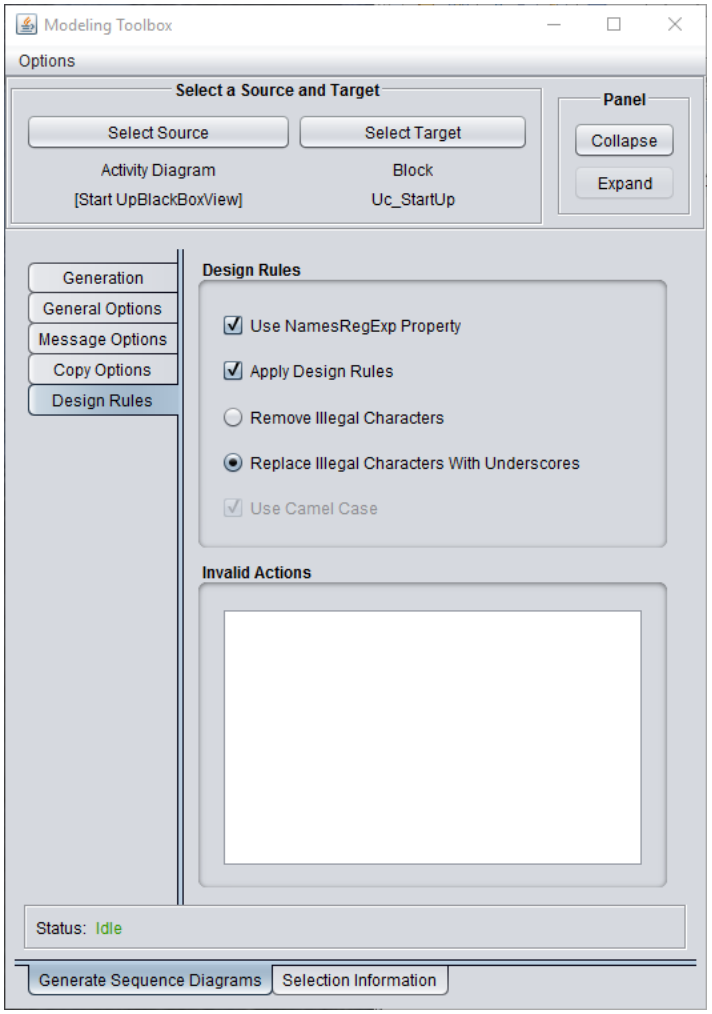


Figure 61: Applying Design Rules

Select *Apply Design Rules* and you can either *Remove Illegal Characters* or *Replace Illegal Characters with Underscores*. If you select the former option, the toolkit will remove the parentheses in the names of the actions. BTW, be sure, under the *Message Optons* tab that *the Use Operations instead of Events* option is **NOT** checked.

Now select the *Generation* tab and hit *Collect Actions*. The Toolkit will proceed to the end of the activity OR until it reaches a branching decision. The decisions will be shown via highlighting on the diagram and the different options are shown in the Select *Next Action* list.

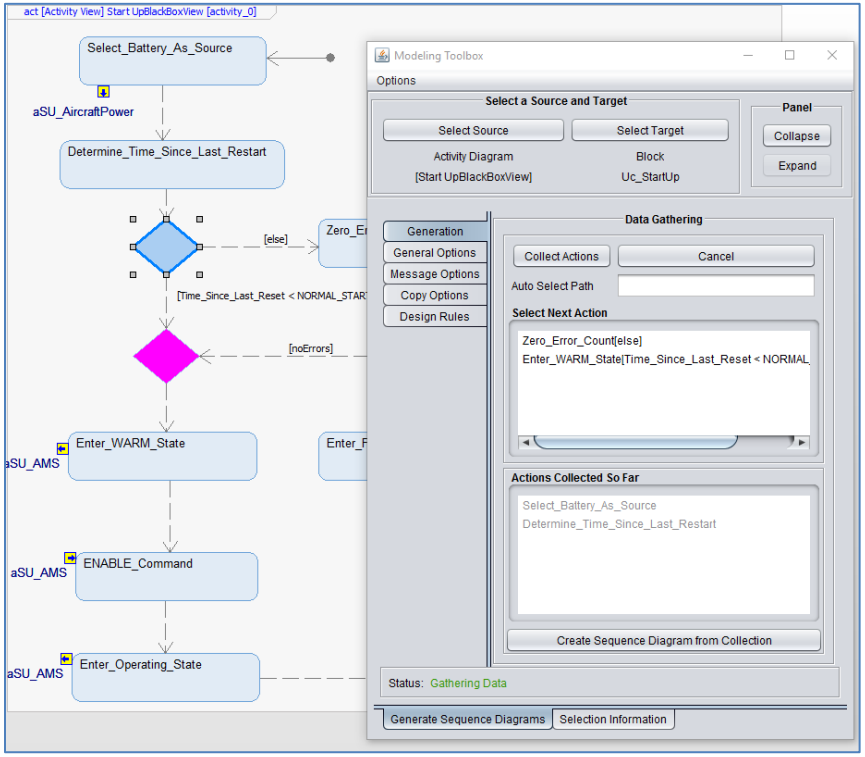


Figure 62: Selecting the path to take

Double click on the **Enter_WARM_state** option. Because there are no more decisions to make, the Toolkit can finish the process and create the entire sequence diagram (Figure 63).

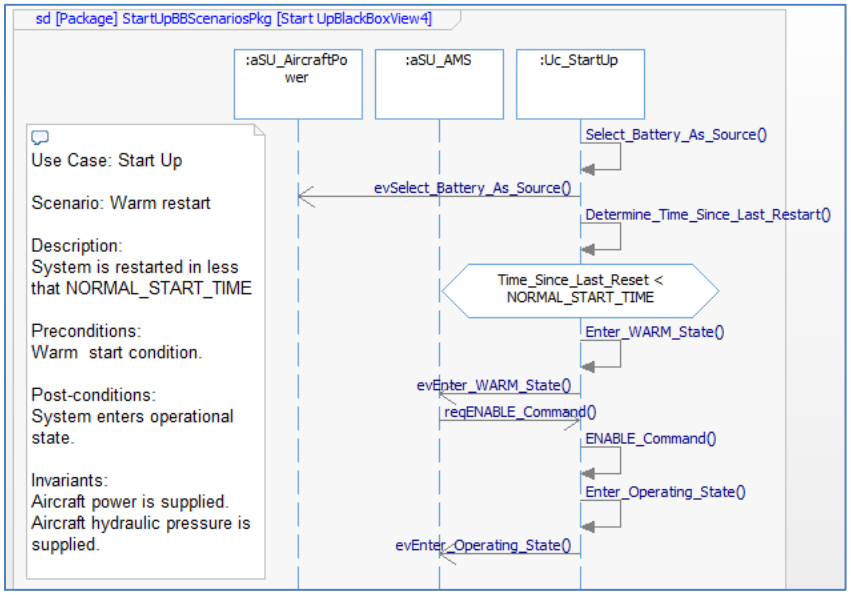


Figure 63: Generated Sequence diagram for warm restart

Be aware that I made two changes to this diagram manually. It is common to annotate and/or elaborate scenarios generated in this fashion. The generated sequences provide most of what you need to capture, but we expect that there will be a small about of manual update to them.

First, I added a comment on the left hand side of the diagram describing the flow. Second, the toolkit replaced "illegal characters" in the condition box so that it read

```
[Time_Since_Last_Restart__NORMAL_RESTART_TIME]
```

I edited that text to put the '<' operator back in place.

```
[Time_Since_Last_Restart < NORMAL_RESTART_TIME]
```

Let's do some more scenarios. We'll need to take the else path when we get to the `[Time_Since_Last_Restart < NORMAL_RESTART_TIME]` decision. This will put us into the **Range_Surface_Test** subactivity (Figure 58). Here there are 4 "interesting"

decisions to make (found various errors or not). For our purposes, it is enough to parse the large loop (decision at the bottom of Figure 58) once. There are also six "interesting" scenarios from the **Perform_BIT** subactivity. Ideally, each decision path would be taken in at least one scenario. To save space in this document, we will do only three more.

1. Range surface and POST tests all pass
2. Range test fails minimum position test but passes all other tests
3. Maximum range test fails and SW integrity test fails

If the Modeling Toolbox isn't already open, right-click again in the main activity diagram and select *SE-Toolkit > Generate Sequence Diagrams*. Make sure the design rules are set, and then click on *Collect Actions*. Double click the **Zero_Error_Count(else)** path and continue to generate the sequence diagrams.

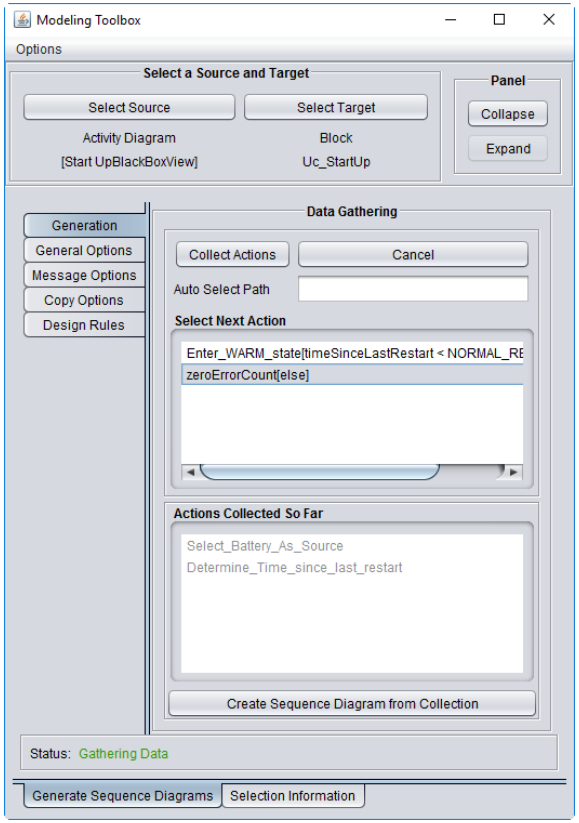


Figure 64: Selecting the else path

For the current sequence diagram, select the else paths (no errors) until you're back at the main diagram, then double click on the **noErrors** path to get to the **WARM** state. From there, there are no more decision points, so the tool will complete the generation of the sequence diagram (Figure 65).

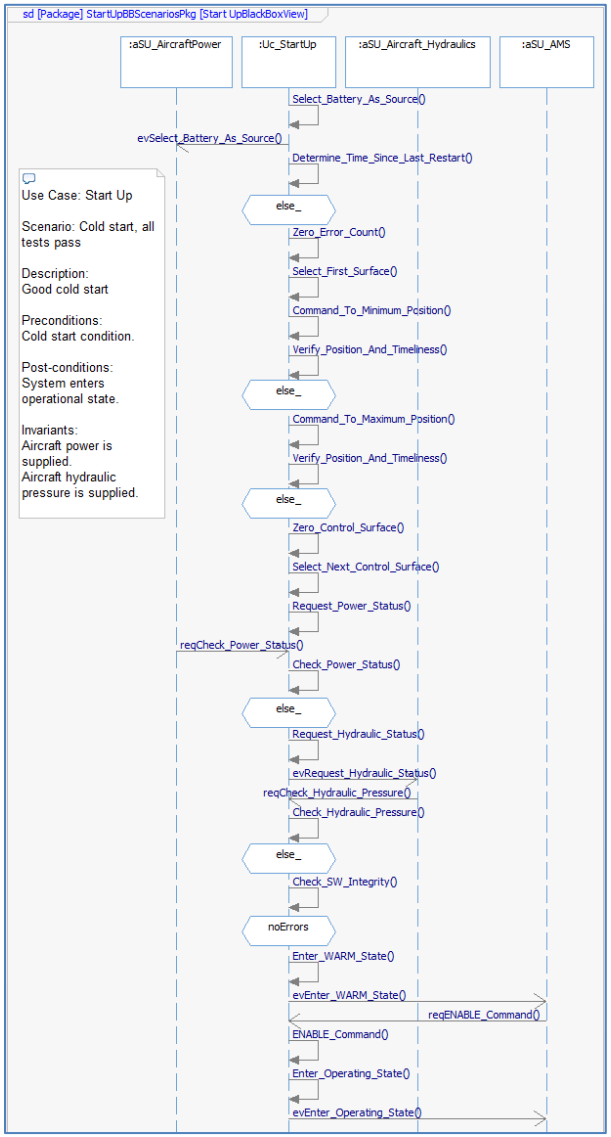


Figure 65: Scenario 2: Cold Start All Tests Pass

Note that toolkit modifies the messages associated with the actor pins to become events to or from that actor on the sequence diagram.

Generating the other cases is straight-forward. The scenario for case 2 “Minimum range test fails but all other tests pass is shown in Figure 66.

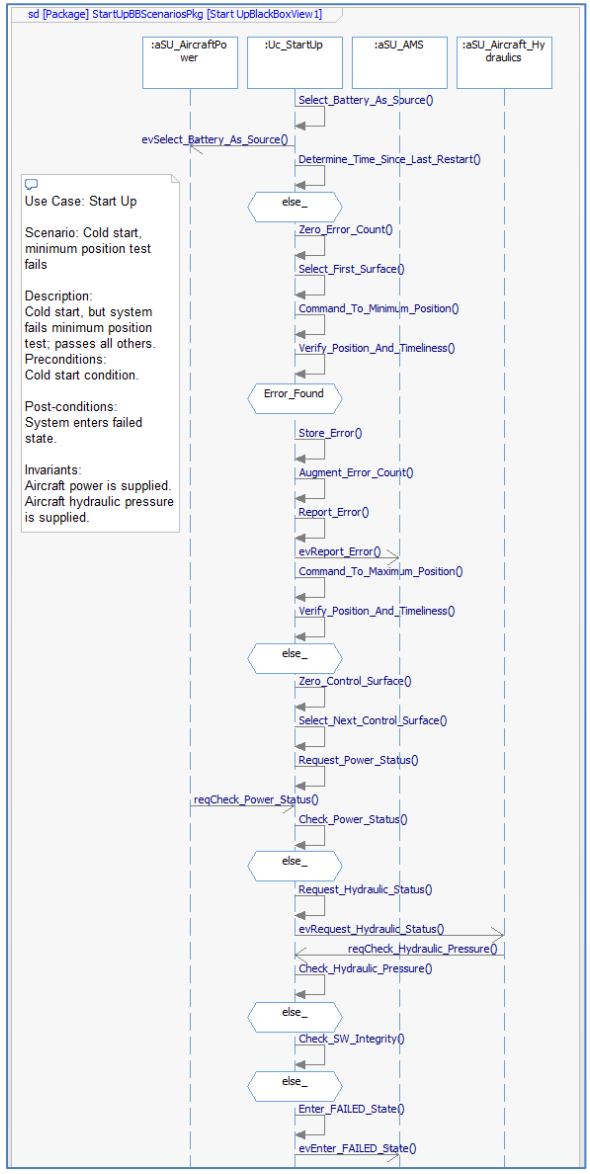


Figure 66: Scenario where minimum range test fails

Lastly, we'll generate the longest scenario. In this scenario, the maximum position test fails and the SW integrity test fails as well (all other tests pass). Because of the length of this scenario, it is shown in the next two figures, Figure 67 and Figure 68.

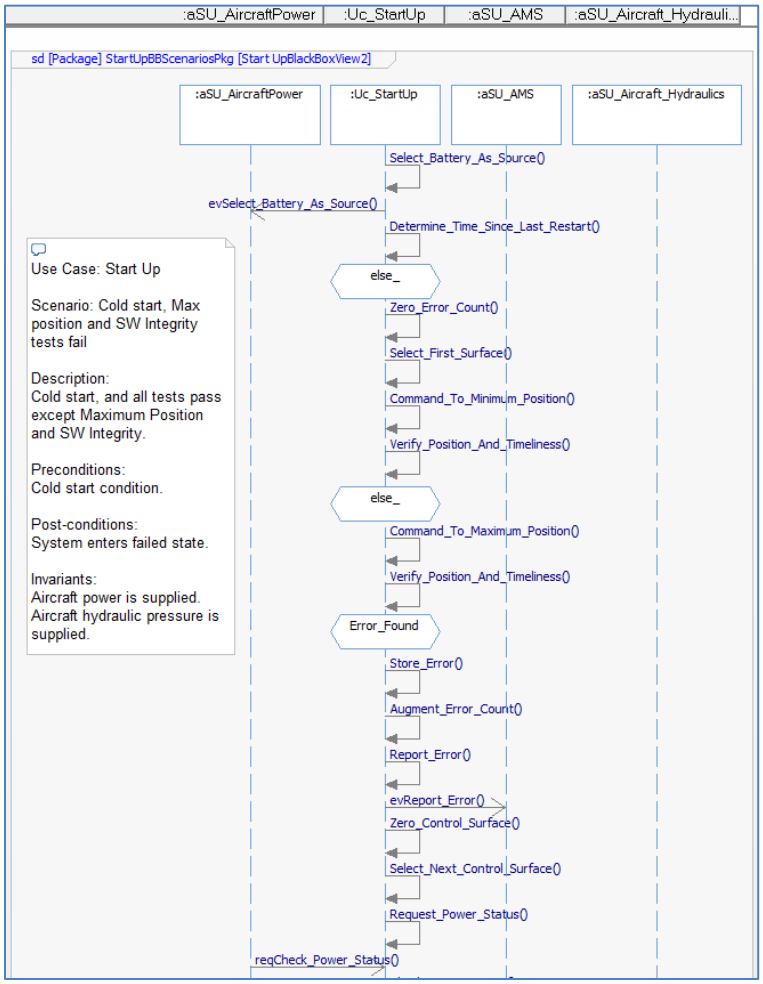


Figure 67: Scenario multiple errors (part 1)

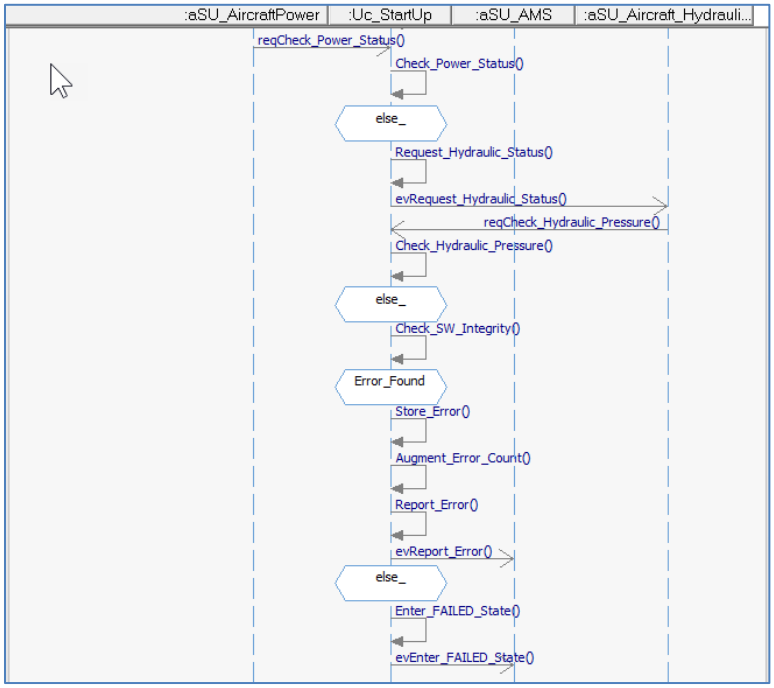


Figure 68: Scenario multiple errors (part 2)

Update the Interface Blocks to include the Events and Flows

Now that the scenarios are done, we can use the *Ports and Interfaces* tool to add the events (generated along with the sequence diagram), to the interface blocks. The toolkit created events for the messages between the actor blocks and the use case block, using the actor pins as a guide. We will have to modify them later to add data for them to carry but for now, we can go ahead and add these to the actor blocks and the interface blocks.

In the browser, right click on the package **StartUpBBScenariosPkg** and select *SE-Toolkit > Ports and Interfaces > Create Ports and Interfaces*. This will add the events as directed features to the interfaces (Figure 69). Figure 70 shows the feaures added to the actor blocks during the earlier sequence diagram generation.

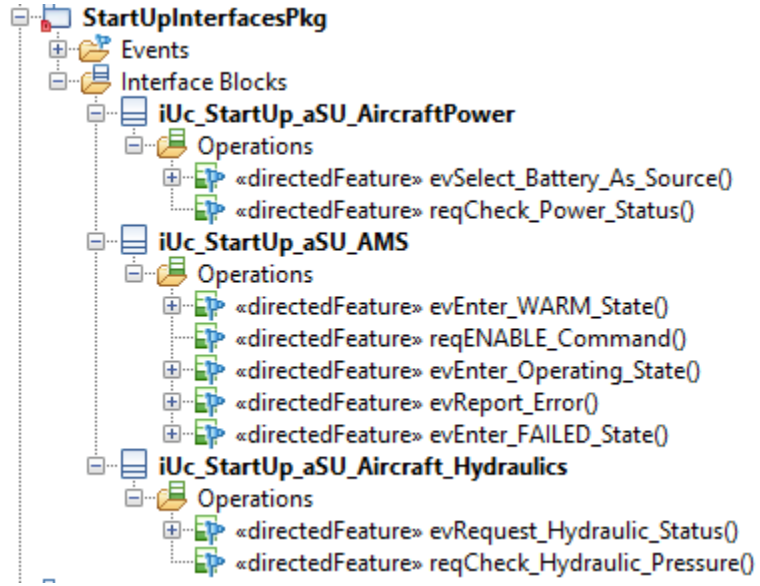


Figure 69: Events added to Interface Blocks

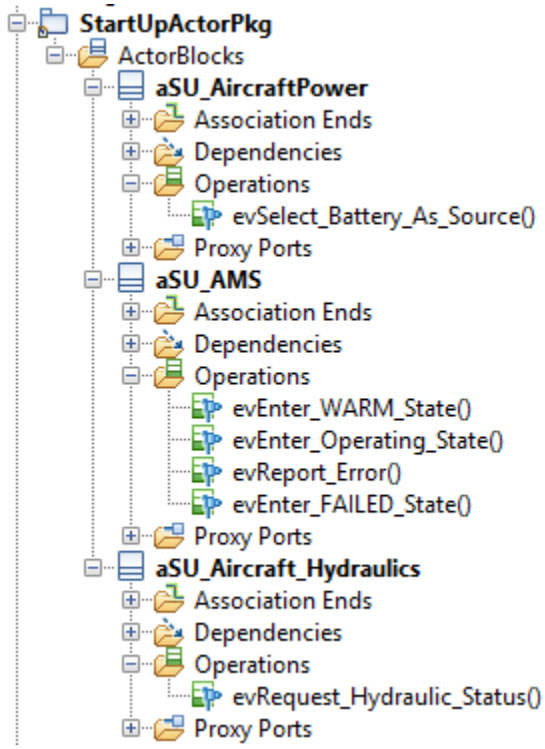


Figure 70: Event Receptions added to the Actor Blocks

7.3.4 Create the Logical Data and Flow Model

The previous steps have identified some flows between the actors and the system while executing the use case and added these as events. More are likely to be identified as we proceed. It is important to note that while some events are data-less, such as the **evEnter_WARM_state** and **evRequest_Hydraulic_Status**. Others need to pass information, such as **evReport_Error** and the poorly-named **reqCheck_Hydraulic_Pressure**. We must create a logical data schema to describe this information and add this information to the events, as appropriate.

The SE-Toolkit uses an automatic naming schema to name the events it generates. The **Check_Hydraulic_Pressure** action is marked with an incoming actor pin from the **aSU_Aircraft_Hydraulics** actor block. The

toolkit assumes this must be a request but really it is a response from a query. Let's rename the event to **herezaHydraulic_Pressure**[5]. Do this in the browser. The event is located in the **StartUpInterfacesPkg**. In the browser, select the event and then click again to change the name (or alternatively, double click and do this in the *Features* dialog for the event). While you're at it, change the name of the **reqCheck_Power_Status** to **herezaPower_Status** event.

Rhapsody will retain all the relations to the various messages automatically. If you look at the features of the interface blocks and the actor blocks, you will see that the event receptions are renamed as well. Likewise, the messages on the sequence diagrams are renamed. That's one of the advantages of using a modeling tool.

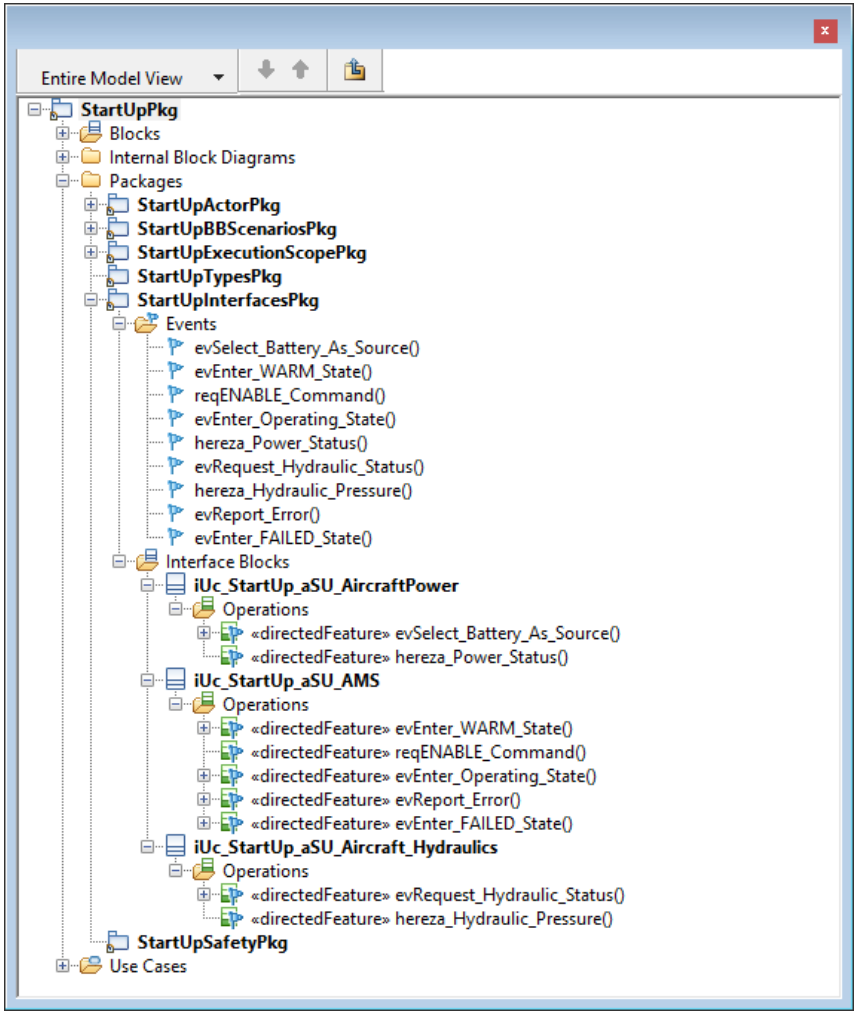


Figure 71: Renaming some events

Now let's model the data.

When we created the functional analysis package structure for the **Start Up** use case with the *Generate System Model from Use Case* tool, a subpackage was created for this purpose. It is the **StartUpTypesPkg** package.

[5] As is "here's a Hydraulic Pressure".

Right click on this package and select *Add New > Diagrams > Block Definition Diagram.* Name this diagram **Start Up Data Schema**. We will enter our types and blocks into this diagram.

We are going to want to see the value properties of the blocks. To do this for the elements we are about to enter, Right click on the diagram and select *Display Options*. Here, click on *Compartment* pane and the *Customize* button to add *EnumerationLiteral* to the compartments displayed. Make sure the *All* radio button is selected and hit *OK*. Now when blocks are added, these visual properties will be used.

Let's think about what information should be returned with an **evReport_Error** event. It makes sense that the AMS would want to know what error occurred, when it occurred and either which surface failed (if, indeed, it was a surface fault), or which power source failed (if a power fault). That gives us a block such as

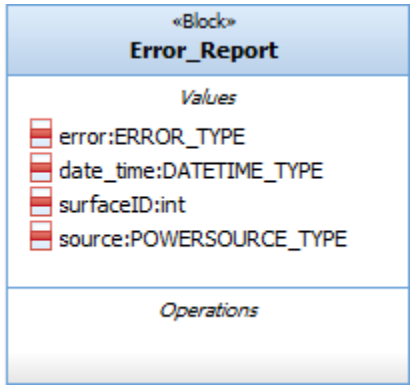


Figure 72: Error Report Type

Let's go about making this type. Let's first define the types of the attributes of the **ErrorReport** type.

We need to characterize the specific attribute types, such as **ERROR_TYPE**, **DATETIME_TYPE** and **POWERSOURCE_TYPE**. The first and the last are best represented as enumerated types. For our purpose, **DATETIME_TYPE** can be represented as a string.

It is important to remember that we are trying to characterize the logical properties of the data and flow – which is why we call this the *logical data and flow schema.* We are not trying to define the final type that will be used in the implementation (this is known as the *physical data and flow schema* and is defined during the hand off to downstream engineering).

Let's create the **ERROR_TYPE** type. In our new BDD, add a *DataType* (alternatively, you can use a *ValueType*) element from the toolbar and name it **ERROR_TYPE**. Double click on it to open its *Features* dialog and in the *General* window pane, set its *Kind* to *Enumeration*. Then click on the *Literals* pane and enter the following values:

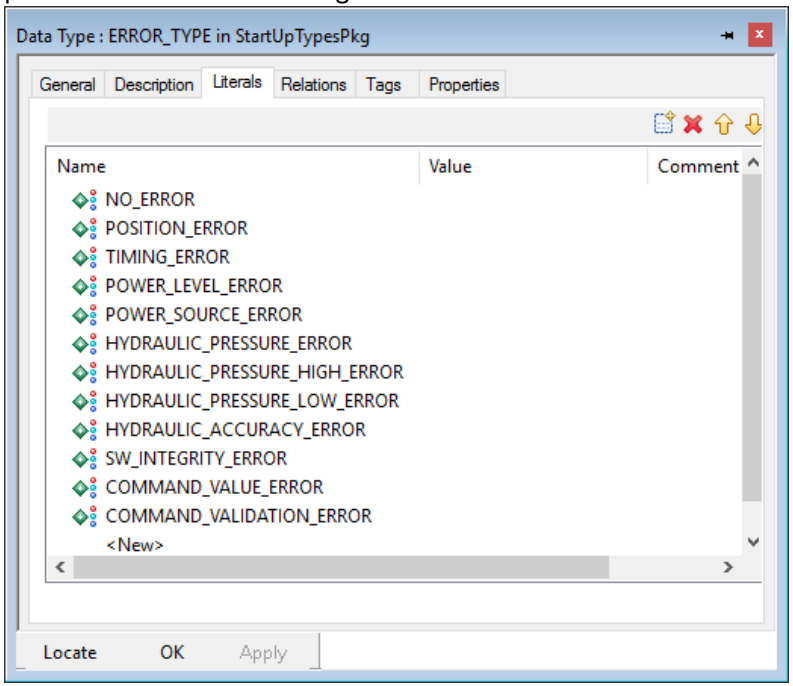


Figure 73: ERROR_TYPE

The **ERROR_TYPE** element on the diagram will now look like this:

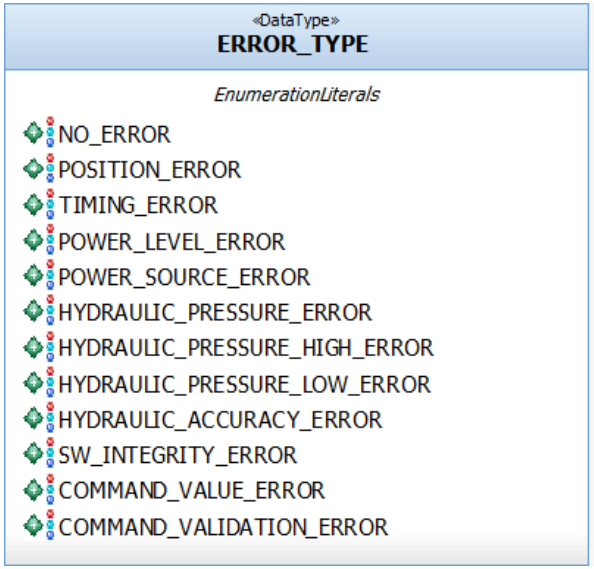


Figure 74: Display of ERROR_TYPE

Similarly, update the **POWERSOURCE _TYPE** with the following literals:

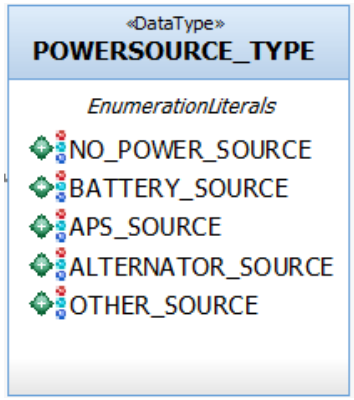


Figure 75: POWERSOURCE_TYPE

Add a new *DataType* or *ValueType* and name it **DATETIME_TYPE**. In its feature dialog, select *Typedef*. In the *Details* window pane, define the base type as *RhpString*.

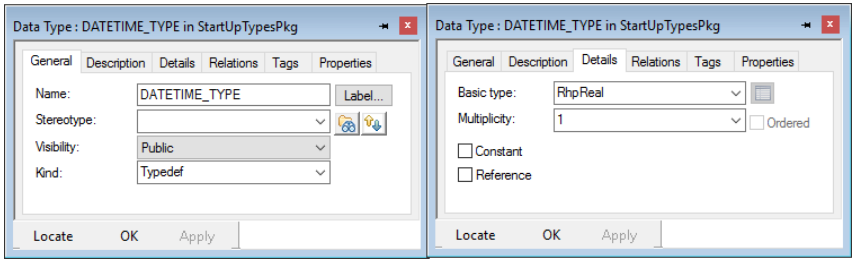


Figure 76: Defining the DATETIME_TYPE as a string

We are now ready to create the **Error_Report** type per se. Add a new block to the diagram and name it **Error_Report**. Double click on the block and click on the window pane *Value Properties*. Add each of the following values, using the *Type* drop down list to select the appropriate types we just created:

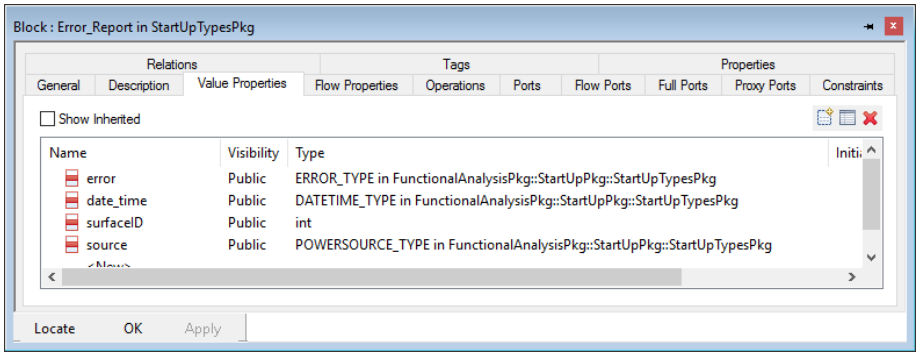


Figure 77: ErrorReport block

We have requirements about keeping a list of identified errors, so add an **Error_Log** block that is composed of zero-or-more ("*") **Error_Reports**.

Other message carry power and hydraulic status, so let's add blocks for those as well. In this case, we're using only predefined types for the attributes, but some of them can be found by navigating to the SysML profile *SIDefinitions* package:

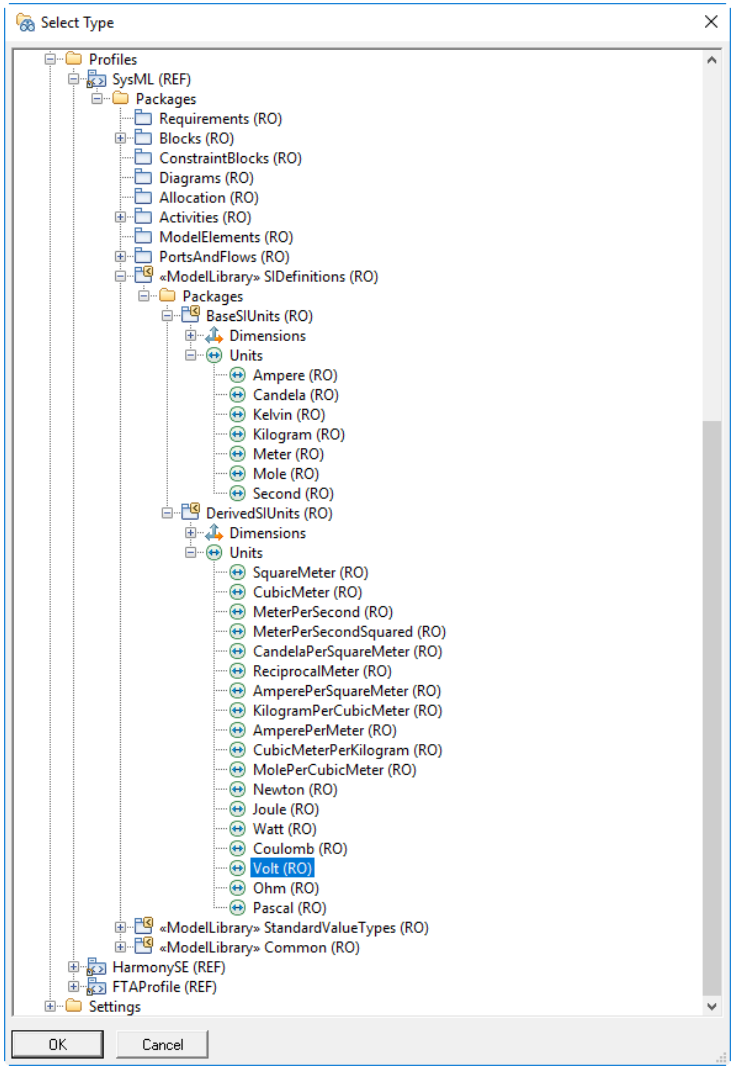


Figure 78: SysML Profile SIDefinitions Package

Using the tool facilities we've already used, add the **Power_Status** and **Hydraulic_Status** blocks to the diagram:

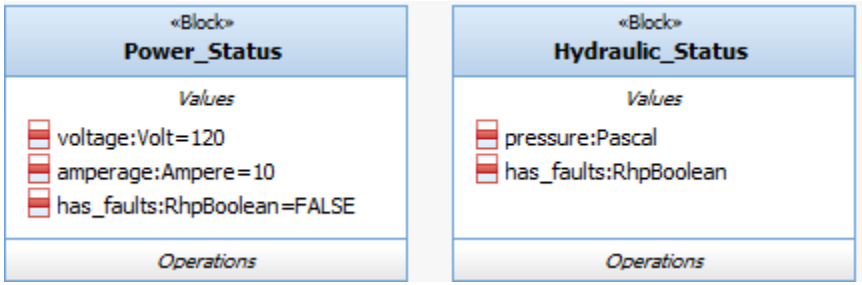


Figure 79: PowerStatus and HydraulicStatus

Note that we assigned the default values to the value properties. This is just good practice and it means that we know the starting conditions when we start simulating. We can do this either on the *Value Properties* tab of the block *Features* dialog or on the *General* tab for the *Features* dialog for the individual value properties.

Lastly, we also have some requirements about storing test results, so define that as well. When completed, the diagram should look something like Figure 80.

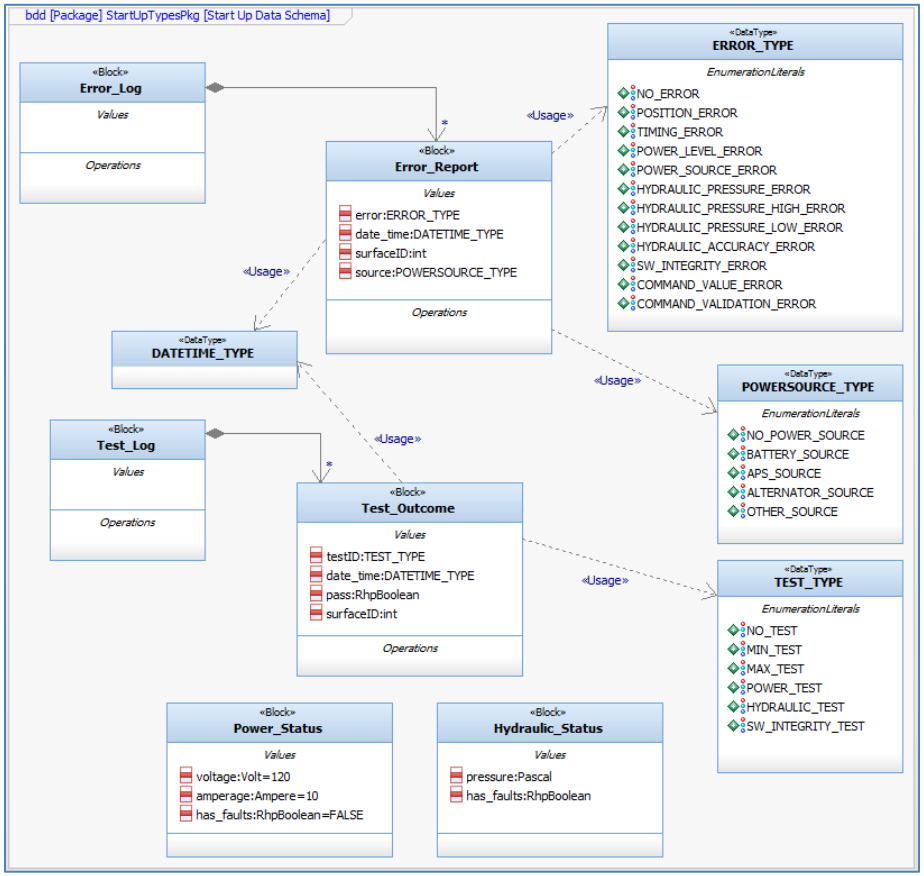


Figure 80: Start Up Use Case Logical Data Schema

I like to use the **«Usage»** dependency between the composite blocks and the type definitions for the attributes, since I find this makes the information more comprehensible. It is, however, optional.

We've now defined the type of interest in this use case (we may find more later but we'll add those as we discover their need). Let's now update the events so that they can pass along that information.

Adding parameters to events is easy in Rhapsody. Open the browser to the **StartUpInterfacesPkg** and click on the plus sign on the events to view the list.

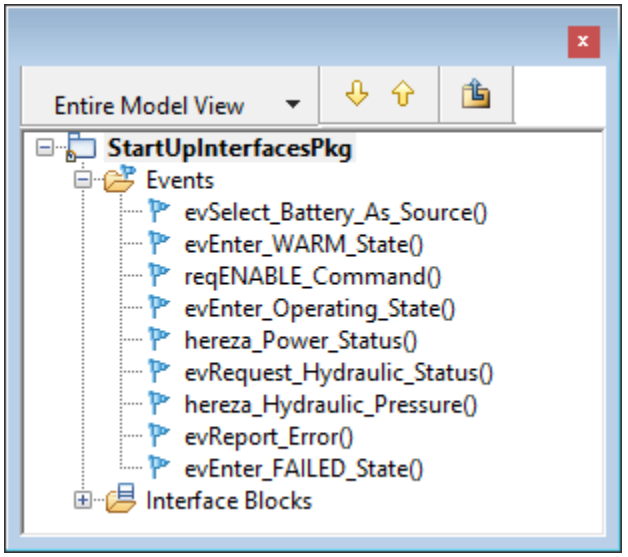


Figure 81: Start Up Use Case events

To add parameters to the **evReport_Error** event, double click on that event to open its *Features* dialog and click on the *Arguments* pane. Here, add an argument **err** of type **Error_Report**.

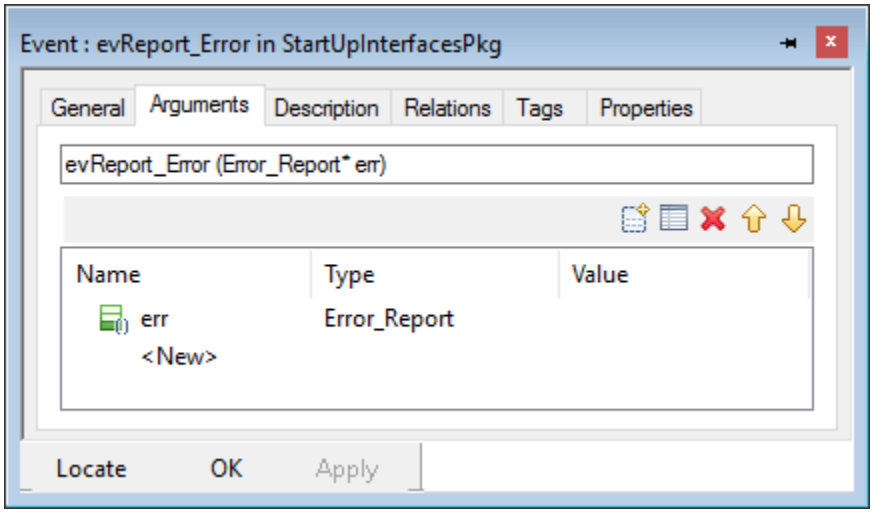


Figure 82: Adding err Argument to evError_Report

Too many entries in the type drop down list?
When you select the *Type* drop down list, you often get a (very) long list of types from which to choose and finding the one you're looking for can be hard.

So here's a Pro Tip:
Start typing the name of the type you're looking for and Rhapsody will shortened the displayed list to just those types that match the partially filled out name.

Using similar methods, add an argument **ps** of type **Power_Status** to event **hereza_Power_Status**, **hs** of type **Hydraulic_Status** to event **hereza_Hydraulic_Pressure** and **source** to event **evRequest_Power_Status**. The browser listing of the events should now look like this:

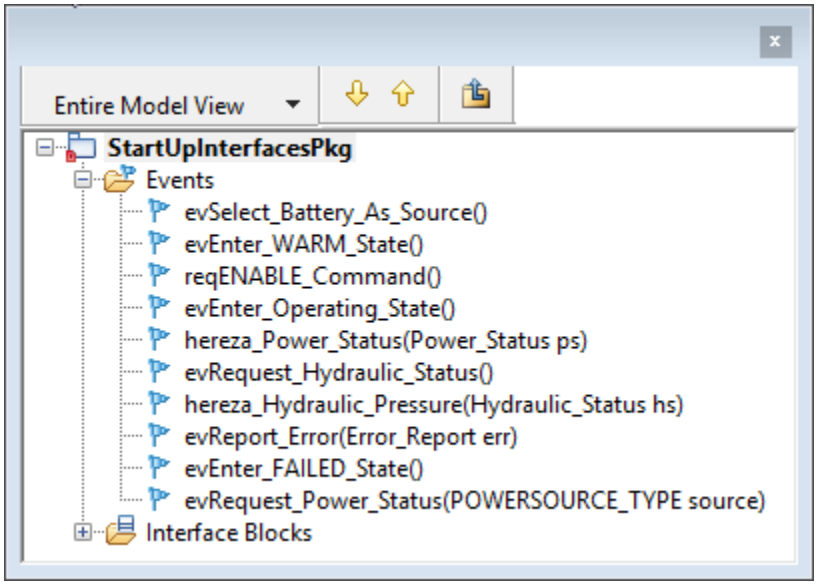


Figure 83: Events updated with arguments

If you want to know more about passing data with events, see **Section 12: Appendix: Passing Data Around in Rhapsody for C++** on page 235.

7.3.5 Create the Safety Analysis

Note: this section is optional. If you never create high-reliability, safety critical, or security-sensitive systems, feel free to skip this section and go on to Section 7.3.6.

Another important source of quality of service requirements are safety requirements. To that end, we will perform a safety analysis of the functional and quality of service requirements on a use base basis.

Installing the Rhapsody Dependability Profile
We will use the Rhapsody Dependability (formerly, the "FTA Profile"). This profile doesn't ship with Rhapsody, so you'll have to download it from Merlin's Cave, where it is part of the Dependability Profile: http://merlinscave.info/Merlins_Cave/Models/Entries/2017/3/3_Dependability_Analysis_Profile.html.

(If you prefer to work in a third party tool, that's fine as well. We'll continue this section assuming you're using the Dependability profile.)

Once you download the zip file, place it in the Rhapsody *Share/Profiles* directory (the same place from which you got the Harmony SE profile) and then unzip it. The proper directories will be created. Then add the Dependability profile in the same way that you added the Harmony SE profile. Inside the **FunctionalAnalysisPkg > StartUpPkg** add a new package **StartUpSafetyPkg** package (the SE Toolkit may have already added this package for you). This package will hold all our safety analysis for the use case.

Note: At the time of this writing, there is an "idiosyncrasy" in the way Rhapsody uses some properties in its API. For this reason, the background of some of the iconic images in FTA diagrams will be red. If you change the type of the project to an *Dependability Profile* type, then that issue is resolved. We recommend that when you're working in the Dependability profile, you change the type of the project to *Dependability Profile* and when you're doing other things in SysML, you change the project back to a *SysML* project.

Changing the project type is easy. In the browser, right click on the project name and select *Change To > Dependability Profile*. To change it back, select *Change To > SysML.*

Doing the Safety Analysis

Let's think about the hazards related to this use case. A *hazard* is a condition that leads to an accident, loss, or incident of concern. In this use case, one hazard is "allowing the pilot to proceed with operations even though the control surfaces cannot be properly controlled." For short, let's call this **Unable to Control Surface**.

Let's be a bit more specific by identifying special cases of this:

- Unable to accurately achieve desired position
- Unable to achieve position within required timeframe
- Unable to power the system
- Unable to move surface
- Operating with faulty software

Any of these conditions could result in the manifestation of the hazard condition. These "sub-conditions" are called *resulting conditions*, because the result from more primitive underlying conditions, events, and faults.

In the context of this use case we're only concerned about safety issues that occur due to or resulting from starting the system up. We are not concerned here about using the system operationally – we'll talk about those concerns when we analyze the **Control Air Surfaces** use case later.

Given that scope, with what functionality must we be concerned? Basically, we must test the system to ensure it is ready to begin operations, and prevent it from going operational if not. This is the basis for the definition of the **Power On Self Test (POST)** functionality. Clearly, the authors of the requirements were thinking about safety when they identified the need for the POST. Our job in this safety analysis is to ensure that those requirements are complete, accurate, and correct with respect to the maintenance of system safety.

Each of the identified resulting conditions that can lead to the hazard are the result of more primitive faults. In this case, these basic faults might be things like:

- Hydraulic pressure failure or leak
- Hydraulic overpressure
- Insufficient or intermittent electrical power
- Fault at the site of the control surface itself causing inability to move accurately enough or fast enough
- Previous installation of invalid software

An FTA diagram graphically represents the logical relations between events and conditions (such as faults) with outcomes (such as resulting conditions or hazards). The logic flow is how we causally connect the elements, and the logical operators (AND, OR, NOT, etc) are how we combine them.

Since we have a number of tests, the way to arrive at the hazardous situation is for BOTH the underlying fault to occur AND the test for that fault fails positively (that is – it gives a positive result (test passed) when the result should have been negative).

Let's create a new FTA diagram to capture our safety analysis. Right click on the **StartupSafetyPkg** package and select *Add New > Safety Analysis Profile > FTA Diagram* (Figure 84). Name the diagram **Start Up FTA**.

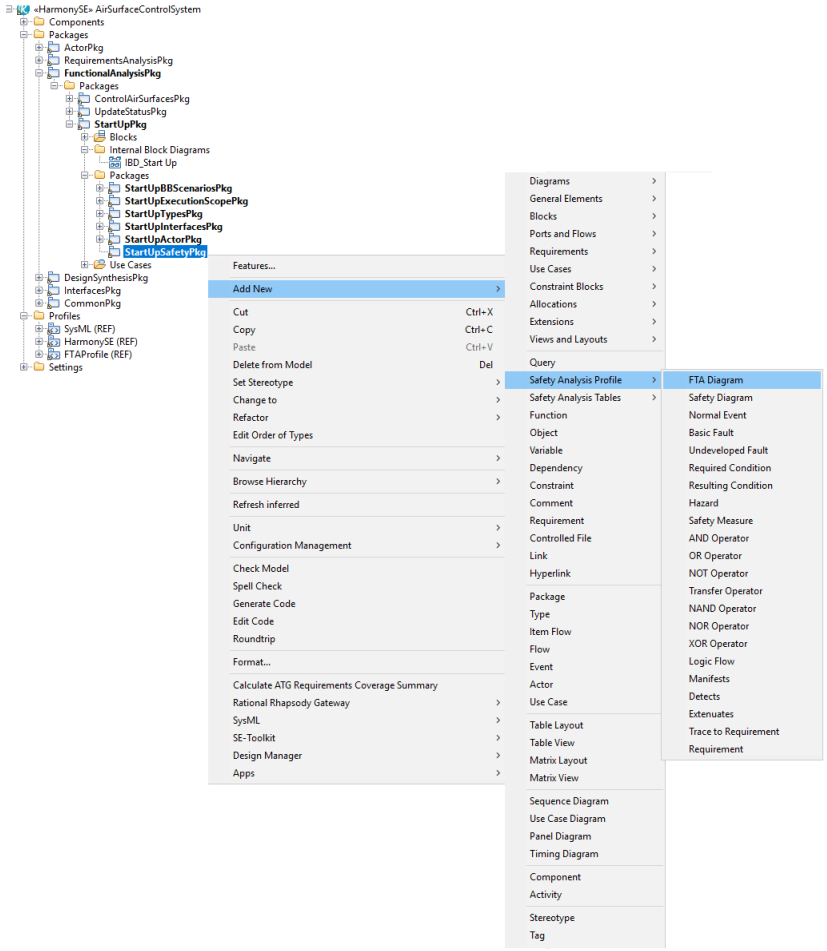


Figure 84: Adding an FTA Diagram

Fill out the analysis in the new diagram by adding the hazards and resulting conditions discussed previously, the logic operators, and the basic faults. The result should look like Figure 85.

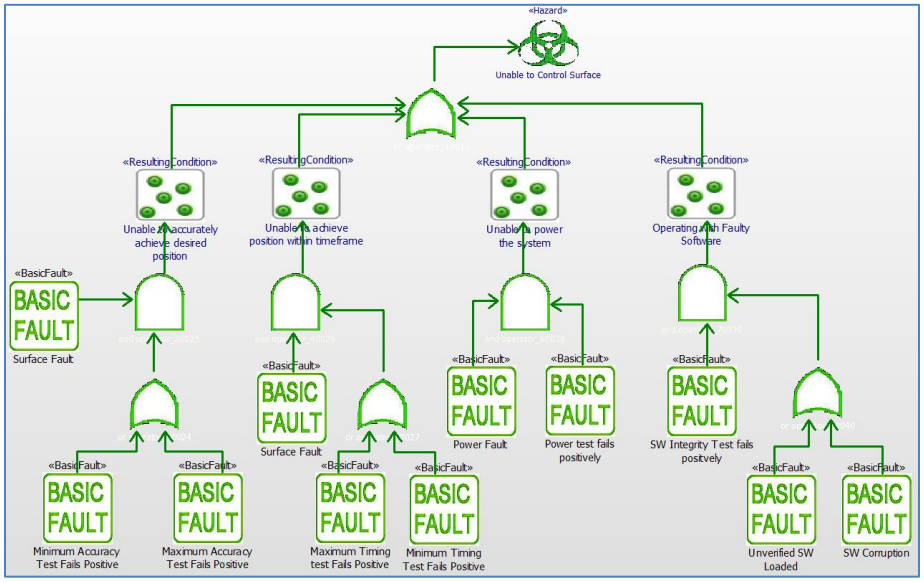


Figure 85: Start Up Use Case FTA Diagram

Figure 85 identifies two ways that the SW Integrity could be faulty. Either an unvalidated software load was performed or the software was corrupted. The existing requirement just calls for a software integrity check but doesn't specify what needed. Here, we need to be able to identify both basic faults. This means with this safety analysis, we've identified the need for three new requirements:

The software load shall provide a key that indicates it has been certified for use.

The system shall verify the software load has been certified by checking the verification key.

The software shall provide a means by which to detect software corruption from initial load, such as a 32-bit CRC check over its contents.

These requirements must now be added into the requirements set in the **RequirementsAnalysisPkg > RequirementsPkg > ErrorReqs** package and

linked to the use case with the appropriate trace dependency relations to the **Start Up** use case. Note that the Dependability profile has a stereotype *«SafetyRequirement»* to mark such requirements if you like; it is available when the project type is **DependabilityProfile**.

Specifying the Safety Metadata

The diagram is a great aid in understanding, but you also need to specify the underlying safety metadata. All the fault and hazard elements have tags to specify this information.

Here's a quick list of the metadata you can specify for the safety relevant elements:

Safety Element	Tag	Description
Hazard	Severity	How bad is an incident resulting from manifestation of this hazard
	Probability	How likely is the accident to manifest?
	Risk	The products of Severity * Probability
	Safety Integrity Level	The level of safety assurance needed – both system- and standard-specific.
	Fault Tolerance Time	How long can a fault be tolerated before the hazard manifests into an incident?
	Fault Tolerance Time Unit	The time unit for Fault Tolerance Time
Basic Fault Undeveloped Fault Resulting Condition	Probability	How likely is the fault to occur?
	MTBF Time Units	The time units for MTBF
	MTBF	The Mean Time Between Failure
	Action Taken	What does the system do to detect, correct or respond to the fault?
	Cause	The underlying cause factor resulting in the fault
	Current Controls	What is in place now to mitigate or control the effect of the fault?
	Detection Mechanism	How the system detects when the fault has occurred?
	Effect	The real-world outcome(s) should the fault occur
	Failure Mode	The mode or ways in which a system or element might fail
	System Function	A behavior of a system which is atomic at a system black box level
	Recommended Action	What are recommendations for additional behaviors for fault control?
	Responsible Party	Which engineer, role, or party is responsible to address the fault?
	Risk Priority	The product of likelihood, severity, criticality and detectability
	Severity	How bad are the outcomes from this fault?
Fault Source	Fault Mechanism	How does the fault happen?
Normal Event Required Condition	Probability	Likelihood of occurrence
Safety Measure	Fault Detection Time	How long to detect the fault after it occurs?
	Fault Time Units	Time units for fault detection and action times
	Fault Action Time	Once a fault action is initiated, how long until it is complete?
	Safety Mechanism	How does the safety control work to mitigate risk?
	SIL	Safety Integrity Level – this is safety standard-specific
Hazardous Event	Probability	Likelihood of occurrence
	ASIL	Automotive Safety Integrity Level – this is standard to the ISO 26262 standard
	ASIL Controllability	How well can the fault event be mitigated?
	ASIL Prob Exposure	Likelihood of exposure of the system to the fault
	ASIL Severity	How bad is an event resulting from manifestation of this hazardous event?
	Effect of Failure	Outcome of the fault

Figure 86: Safety and Reliability Metadata

We can fill in some of this metadata later. For now, let's fill out the fault tree analysis.

By the way, after you're done with the safety analysis, don't forget to change the project back to a SysML project by right clicking on the project name at the top of the browser and selecting *Change To > SysML*.

7.3.6 Create the Use Case State Machine and Execute Model

Next, we want to construct an executable version of the use case model so that we can make sure that the requirements result in the outcomes we want and expect. To do this, we will construct a state machine that takes inputs from the actor blocks, executes internal system functions, and sends output to those actor blocks. This is a black box simulation so it neither reflects the actual internal design nor actually performs the internal system functions needed to actually do anything "real". Our goal is to cast the informally stated textual requirements into the formal language of state machines and run various event sequences through to ensure that we have a correct and complete set of requirements. If we discover inadequacies in the requirements, we update the requirements and our model, and repeat. Again, *the state machine used here is just a statement of the requirements in a more formal language, not a specification of internal design*.

- By the way, did you remember to change the project back to a SysML project?

A Note about Simulation Fidelity

Simulation can be done at different levels of detail, known as "fidelity". These simulation levels have both benefits and costs.

A low-fidelity simulation can be done by executing the data machine with no event parameters and keeping the data model and the behavioral model separate. This approach was taken for "Harmony Classic" and can still be used, if desired. This level of fidelity does allow the verification of the control flow of the use case but not the correctness of the data model. While this simplifies the work to get the simulation running, you still have to add the data elements to the interfaces later, because they are a very important part of the specification.

A medium-fidelity simulation models the logical data passed by the events. It's a bit more work to get the simulation working but the executing state machine relies on the actual logical interfaces, so this verifies, through execution, the correctness of these logical interfaces and the data they support. *This is the level of fidelity we will use in this Deskbook.*

In contrast, a high-fidelity simulation also models the internal behaviors and algorithms. This is useful in architecture and design, but less so in requirements analysis. This is level of fidelity requires the most work on the part of the engineer but allows for the effectiveness of design decisions to be ascertained.

Create the State Machine

Correctly constructing complex state machines is hard. Therefore, we will construct this state machine in three phases ("nanocycles" in Harmony-speak). Each phase will be executed before moving on. This incremental construction of potentially complex state machines is highly recommended.

Ideally, a complex state machine should be constructed by representing a small set of requirements and executing it after no more than an hour of

development. Only after that simple state machine verifiably works should the engineer move on to elaborating the state machine with more requirements. In reality, a state machine of this complexity would probably be constructed with 6-10 iterations but we will only show three here to conserve space. Again, the recommended workflow is simple and iterative:

- Model a few requirements in the state machine
  - Get this model to compile and run
  - Verify that it is correct so far
- Repeat until done

Phase 1: Overall state machine

In the browser, navigate to the use case block at *FunctionalAnalysisPkg > StartUpPkg > Blocks*, right click on the block **Uc_StartUp**, and select *Add New > Diagrams > Statechart*. Then create the following state machine (Figure 87):

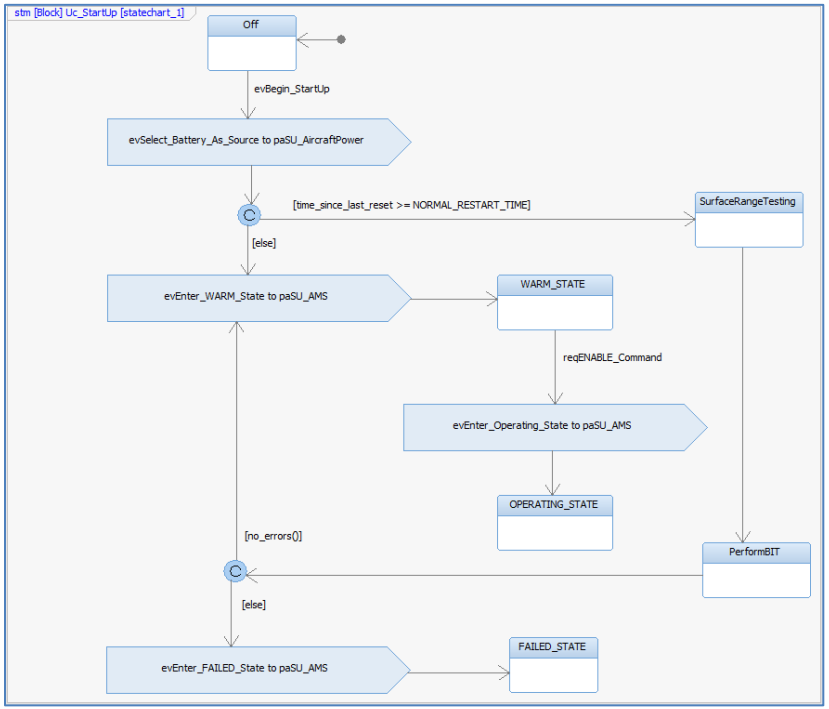
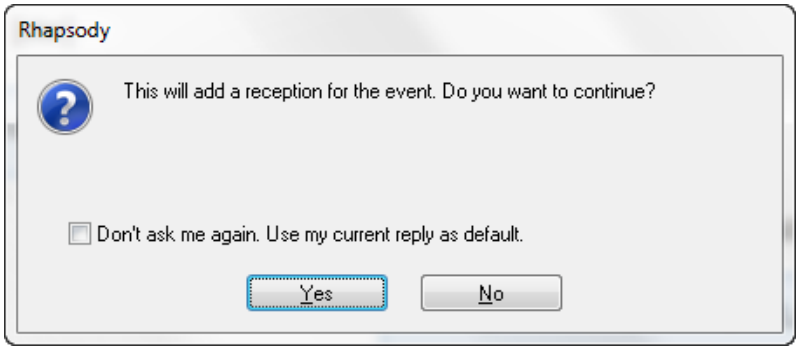


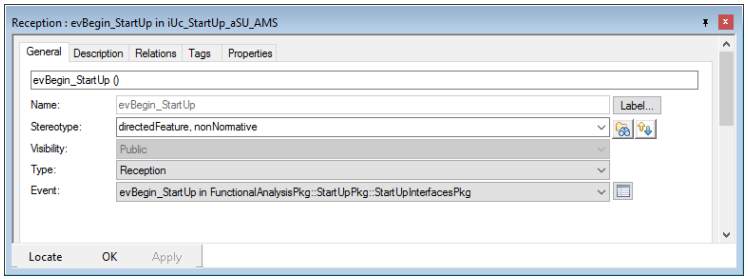
Figure 87: Start Up Use Case State Machine Phase 1

Notice that we added a new event **evBegin_Startup** (we did this for simulation control reasons, so we should stereotype it as «nonNormative»). We defined a state **OFF** and the event **evBegin_Startup** invokes a transition to get things started. This will end up coming from the **aSU_AMS** actor block (sitting in for the **AMS** actor). The event **evBegin_Startup** must be manually added to the interface block **iUc_StartUp_aSU_AMS** as a directed feature with the direction of *provided*.

One easy way to do this is to select the event in the browser (it's in the *StartUpPkg > Events* list) and drag it with the *control* key pressed to name in the browser of the **iUc_StartUp_aSU_AMS** interface. Then a dialog will pop up asking if you want to add an event reception for this event to the interface block.



Click on *Yes*. Then double click on the event reception in the interface block to add the stereotype *directedFeature*.



Since the default direction is *provided*, that's all we need to do.

The other event on the diagram **evRequest_Enable** is located in the nested **StartUpInterfacesPkg**. The easy way to find it is to, when entering the name of the event, type a few letters, such as "**req**" and then press Control-Space to bring up the *intellisense* editor. If the desired event isn't shown, double click on the *Select* option that pops up and navigate the mini-browser to fine the desired event and click on *OK*.

To enter the *Send Action* pseudostates on the diagram, add the *Send* pseudostate on the diagram and double click on it. Then you can fill out the *Target* and *Event* fields from the drop down lists.

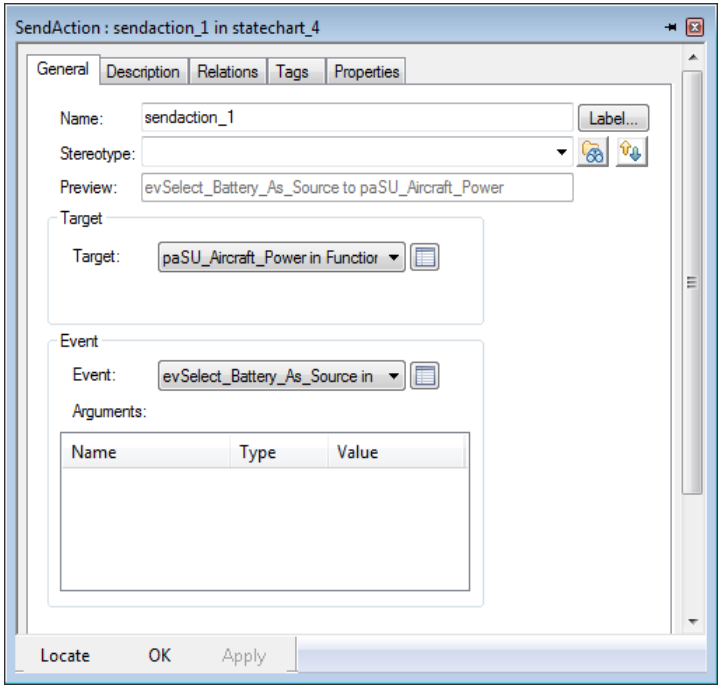


Figure 88: Filling in the details for a Send action

There are a number of minor things we need to do to get this to compile and execute.

Two values are referenced in a guard in Figure 87. **time_since_last_reset** will need to be defined (and initialized) as a value property (attribute) of the **Uc_StartUp** block. The other value, **NORMAL_RESTART_TIME,** we will define as a constant.

There is the use of a function **no_errors()** that must be added as an operation to the **Uc_StartUp** block.

After that, we'll need to add state behavior to the actors, to send and receive the events during the simulation.

Let's begin by adding the value property **time_since_last_reset**. In the browser, right click the **Uc_StartUp** block and select *Add New > Blocks > Value Property*. Give it the name **time_since_last_reset**. The default type (*int*) is ok. Since most of the time we want to execute the start up tests, let's set it to a large value, 100,000. Double click on the **timeSinceLastReset** value property[6] in the browser and enter this value as the *Initial Value*. Click on *OK*.

[6] Attributes in UML are known as Value Properties in SysML. Sometimes what you expect to be a value property will appear as a attribute.

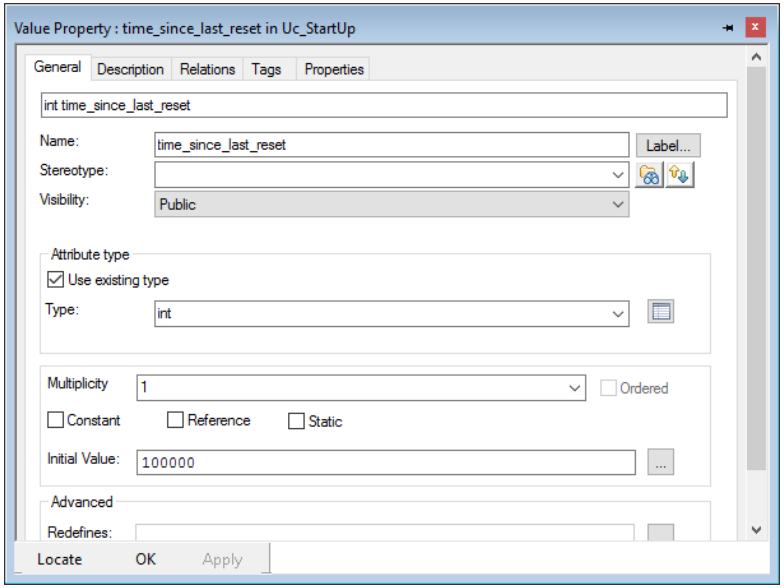
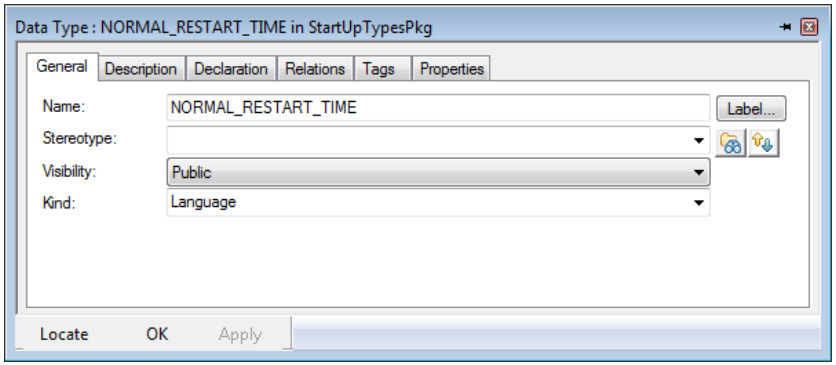


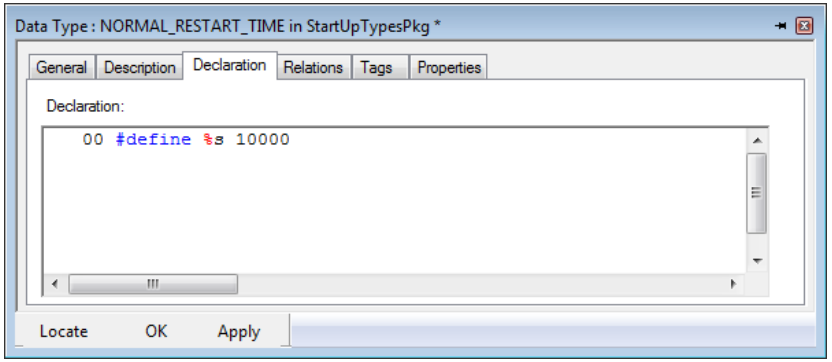
Figure 89: Setting the initial value of an attribute

To define the constant **NORMAL_RESTART_TIME**, right-click on the nested **StartUpTypePkg**, and select *Add New > Blocks > DataType*. Double click on the type to open the features dialog and type in the name of the value. Make sure the *Kind* is *Language*.



Click on the *Declaration* tab and type

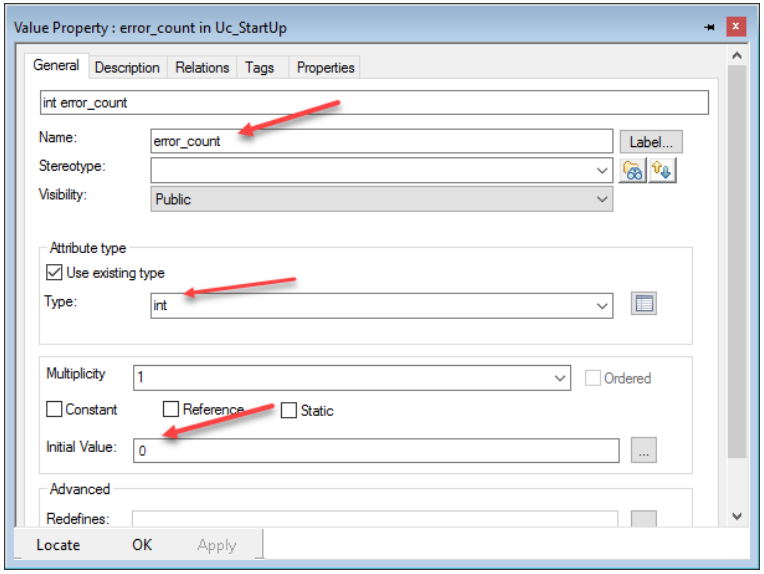
```
#define %s 10000
```



This value is less than the true value (5 minutes) as 10,000 represents only 10 seconds (timeout units in Rhapsody are milliseconds). The actual value is a bit cumbersome to use in simulations, so we'll employ this shorter value. To get a warm restart we only must set the **time_since_last_reset** to less than 10,000. The use of named constants like this makes the model more readable and easier to customize for different simulation effects.

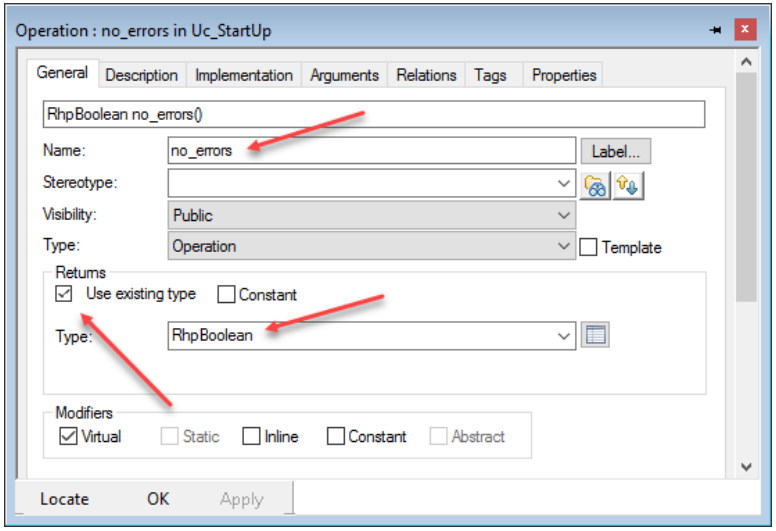
Adding the operation **no_errors()** is likewise easy. Let's do this by
1. Adding an **error_count** attribute/value property and initializing it to zero
2. As we add errors during tests (we'll start doing this in nanocycle phase 2), we'll augment this value
3. **no_errors()** will return **TRUE** if the value of **error_count** is zero.

First, let's add **error_count** as an integer value. Follow the same procedure we used for **time_since_last_reset** but instead give it an initial value of 0.

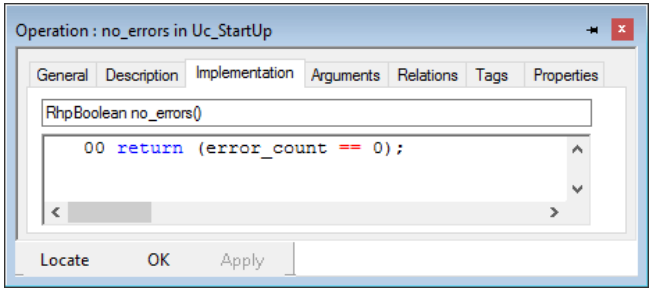


Now, right click on *Uc_StartUp > operations* in the browser and select *Add New > Operation*. Type in the name **noError** and hit the *ENTER* key. Then double click on the operation to open its features dialog.

In the General pane, set the return type to **RhpBoolean**.



Click on the *Implementation* pane and type in the implementation code.



Be sure to use double equals sign ("==") as the operator.

This is all we need to do to the **Uc_StartUp** block for simulation. We still must "instrument the actor blocks" to support the simulation. That means that the actor blocks must be able to send the events through the correct ports and must be able to receive the events from the use case block. To simulate the the initial use case block state machine (Figure 87), we must instrument the **aSU_Aircraft_Power** and **aSU_AMS** actors to accept and receive the events generated by the use case state machine. We need not

do anything (yet) with the **aSU_Aircraft_Hydraulics** as it isn't used (it will be added in Phase 3 of building out this use case state machine).

First, let's do the **aSU_Aircraft_Power**, as this is simple:

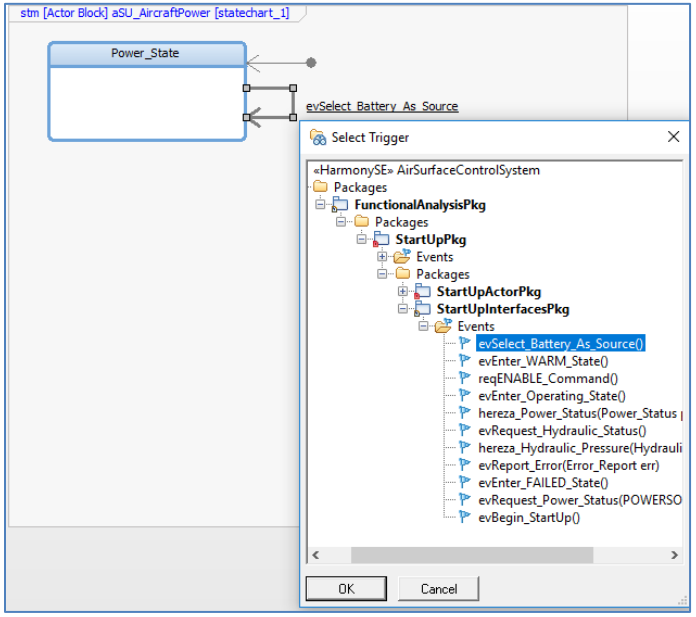


Figure 90: Initial state machine for aSU_Aircraft_Power

This allows us to show the message sent from the system to select the power source. Note that we used the *intellisense* feature (ctrl-space) follow by *Select > StartUpInterfacesPkg > Events* to find and select the event already present in that package. We could have just typed it in as well but in general it is better to select when the event already exists to avoid accidently misspelling and creating a whole new event.

The **aSU_AMS** actor block state machine is a little more complex.

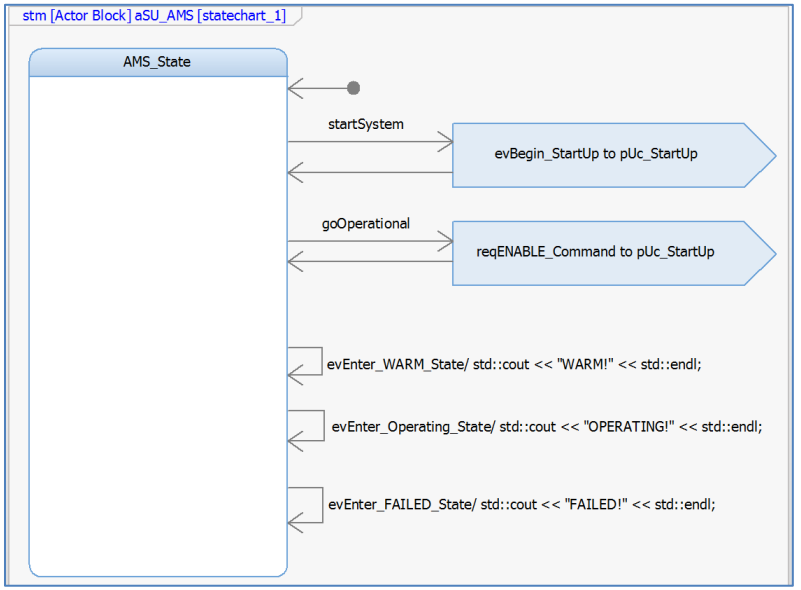


Figure 91: Initial state machine for aSU_AMS

Look at the transition event signatures on the state machine such as

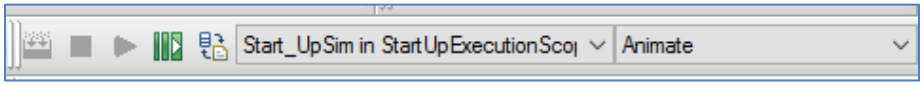
```
evEnter_WARM_state/ std::cout << "WARM!" << std::endl;
```

The action (the part of the event signature following the "/") is optional. Note, by the way, that the `std::` preface for the `cout` and `endl` applicators is required by some compliers (such as Cygwin) but cannot be added in other (such as some versions of the Microsoft C++ compiler). The action list shown just sends the text out to the standard output window for inspection when the event is received. To receive the event, you must minimally have a transition with the triggering event on it; the following action is optional but helpful in debugging.

Adding the behavior to the actor blocks allows us to see the events flowing between the use case and actor blocks during simulation. It also means that we can control execution directly by stimulating the actors.

We are now ready to run this initial state machine. It doesn't do any actual tests yet, but we'll add that behavior in Phase 2 and 3.

Make sure that the simulation toolbar has the (toolkit – generated) component selected.



Navigate to the **StartUpSim** component in the browser; it's located in the *StartUpPkg* > *StartUpExecutionScopePkg*. Double click to open and make sure that the right elements to include are selected in the *Scope* tab.

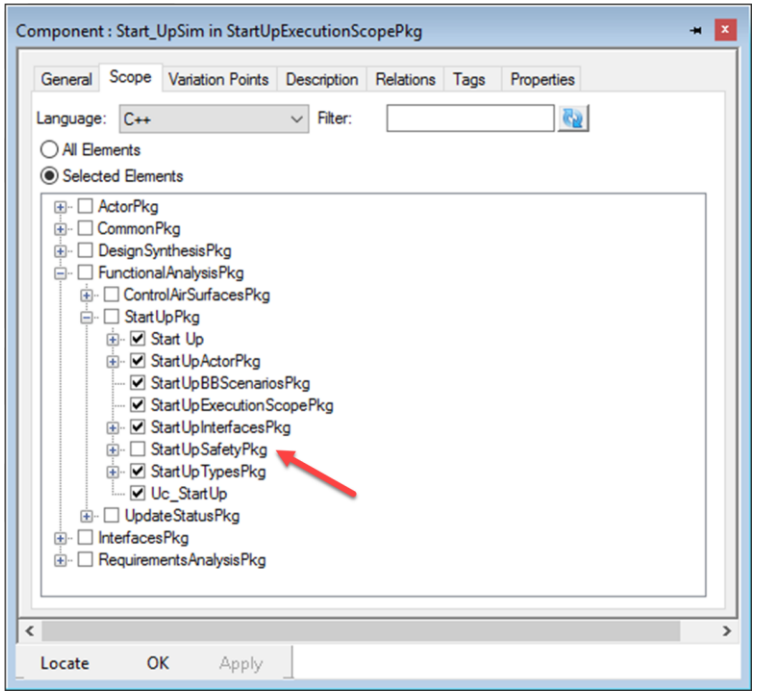


Figure 92: Start Up component execution scope

In particular, note that the **StartUpSafetyPkg** is NOT included in the scope but all the other packages within the **StartUpPkg** are.

Now, verify the configuration is set up for your compiler and for animation:

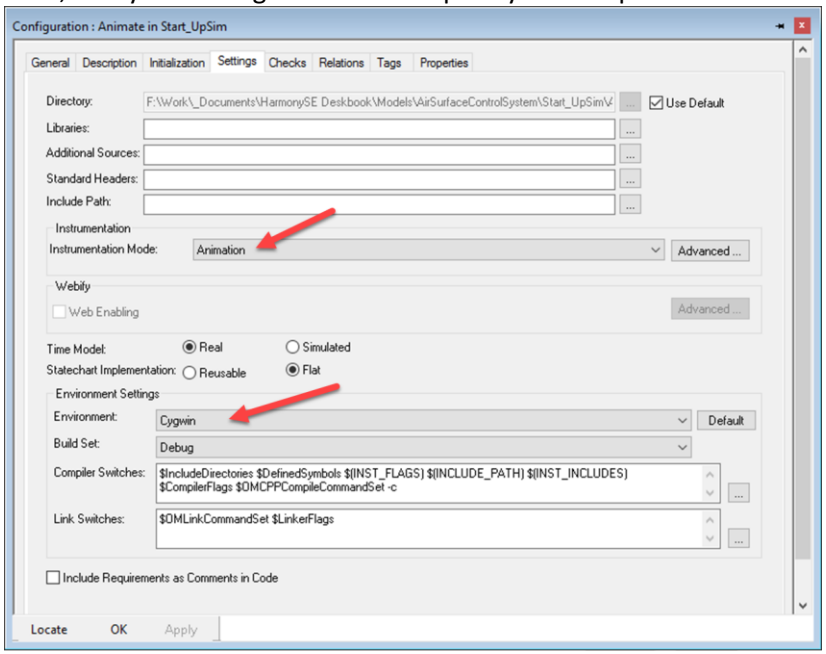


Figure 93: StartUpSim component configuration

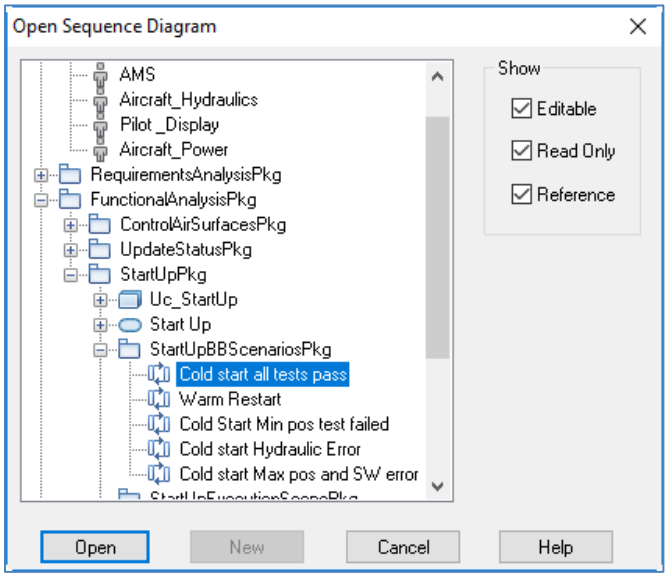
In the case here, I'm using the Cygwin compiler but it must be set up for your particular environment. In any case, you want the *Instrumentation mode* set to *Animation*.

To compile and run the model at this point, just click on the *Generate/Make/Run* button (highlighted in green on the simulation toolbar. If you've make mistakes entering information, the system will stop and identify the error to fix. Don't worry if you've made a mistake. It is so normal that I'm surprised when my compilation succeeds on the first try! The good news is that Rhapsody makes it easy to locate and repair the errors.

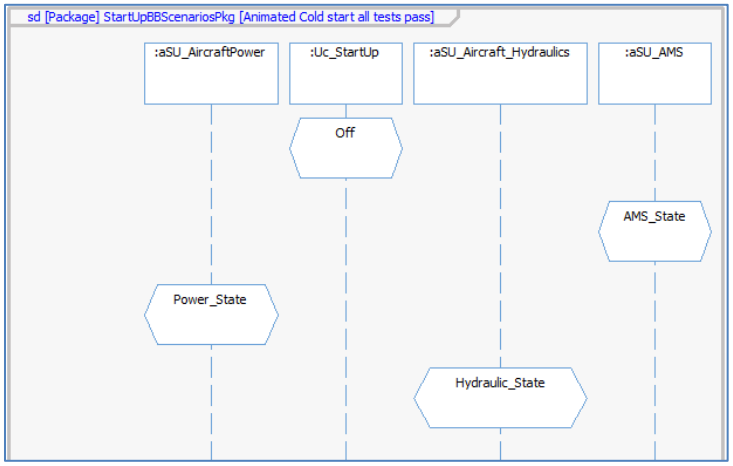
Once the system in in simulation mode, you'll see the animation toolbar:

Click on the *Run Step* tool (third button from the left). This initializes the simulation and the state machine. Let's add an animated sequence diagram to watch the simulation run.

Select *Tools > Animated Sequence Diagram*. Then select one of the diagrams we created previously as the template and click *OK*.



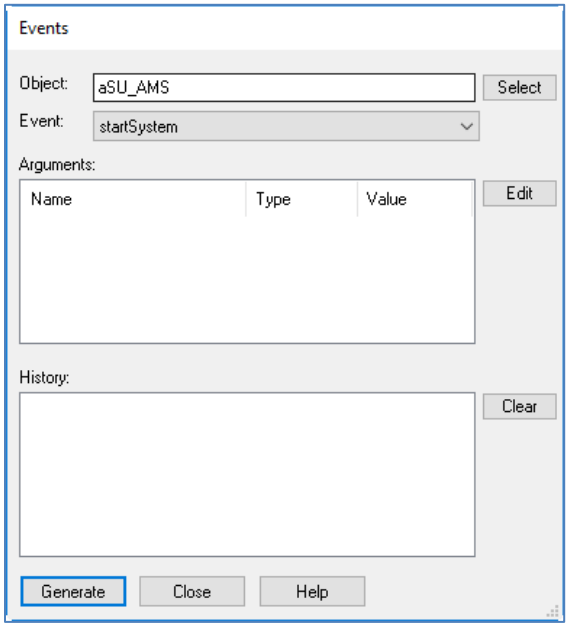
This should open up the diagram and display the current states of the elements.



Now, let's just let the simulation run. Click on the *Go* button on the *Animation* toolbar (second button from the left)[7].

Nothing happens because it's waiting for you to kick things off with an event. On the animated sequence diagram, right-click the **aSU_AMS** lifeline and select *Generate Event*. On the opened *Events* dialog, select the **startSystem** event, and click *Generate*.

[7] If you don't see the State Marks, you can enable the *Features Dialog* by double-cliicking on the project at the top of the browser, selecting the *Properties* Pane, then *View > All,* navigating to *SequenceDiagram > General > ShowAnimStateMark* and selecting the checkbox. Seeing the states is optional but useful.



At this point, we expect the system to run through the **SurfaceRangeTesting** and **PerformBIT** states, and eventually end up in the **WARM_STATE**. And that's what you should see (Figure 94)

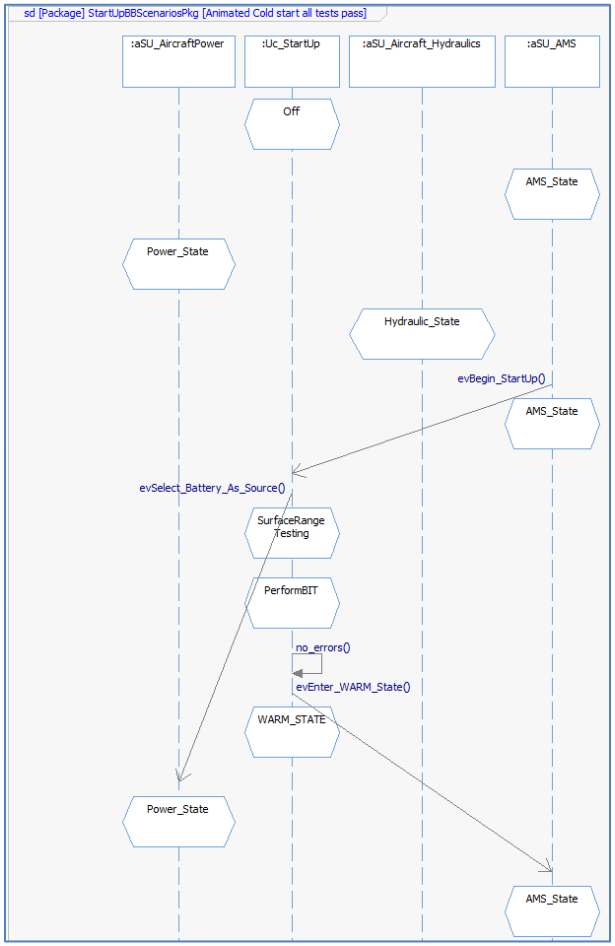


Figure 94: Running the simulation

At this point, using the *Events* Dialog, send the **goOperational** event to the **aSU_AMS** actor block. The system should end up in the **OPERATING_STATE** and the state changes should show up on both the sequence diagram and the standard output window. Alternatively, you can open a *Features* dialog on the animated instance to see its current values.

The sequence diagram looks a bit messy because it marks the arrival of sent events when they are actually processed by the receiving element. However, the SE Toolkit has a great tool to clean this up.

Complete the simulation by pressing the *Stop* button (red square). Then click anywhere in the generated sequence diagram and select *SE Toolkit > Straighten Messages*. Now with a little bit of moving messages up to remove white space, your diagram should look like Figure 95.

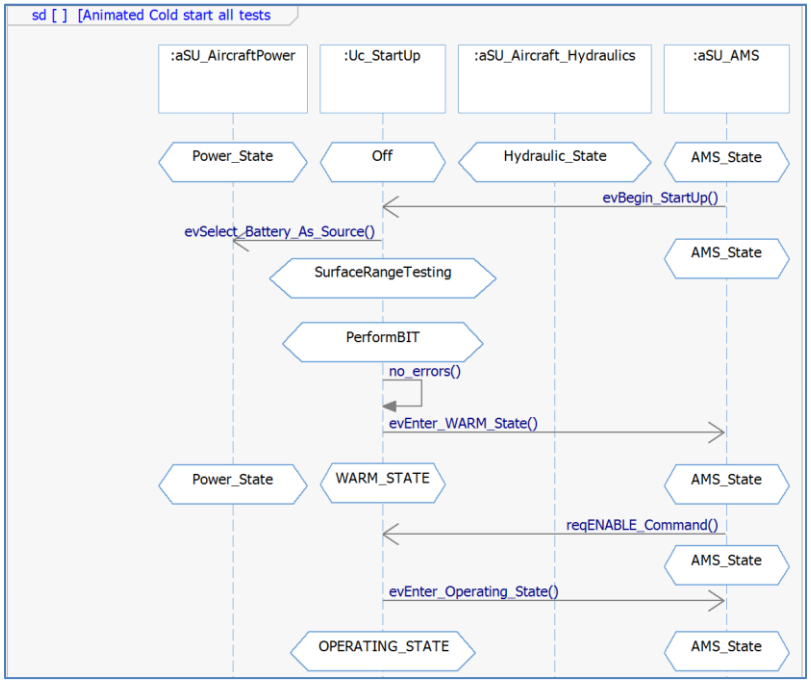


Figure 95: Completed first simulation

To run the other case (warm restart), we'll have to change the values and recompile (in a little bit, I'll show you another way to do this). Stop the simulation. Edit the default value of the **time_since_last_reset** to be a small number, say "10", and now the other main path will be taken. Do this to generate the following sequence diagram:

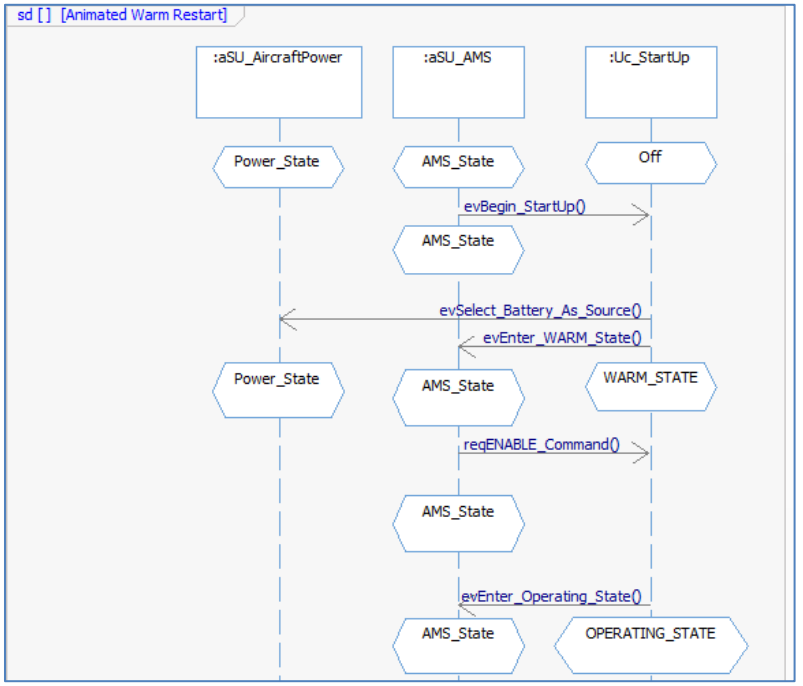


Figure 96: Simulation warm restart

Let's do a final simulation, one where the errors are detected. This is an easy case to do. In this case, stop the simulation, change the default value of **time_since_last_reset** back to it's original 100,000 value and change the default of **error_count** to 1 (thus **no_errors()** will return *FALSE* and the other state machine path will be taken). Then compile and simulate to generate the final sequence diagram of this phase.

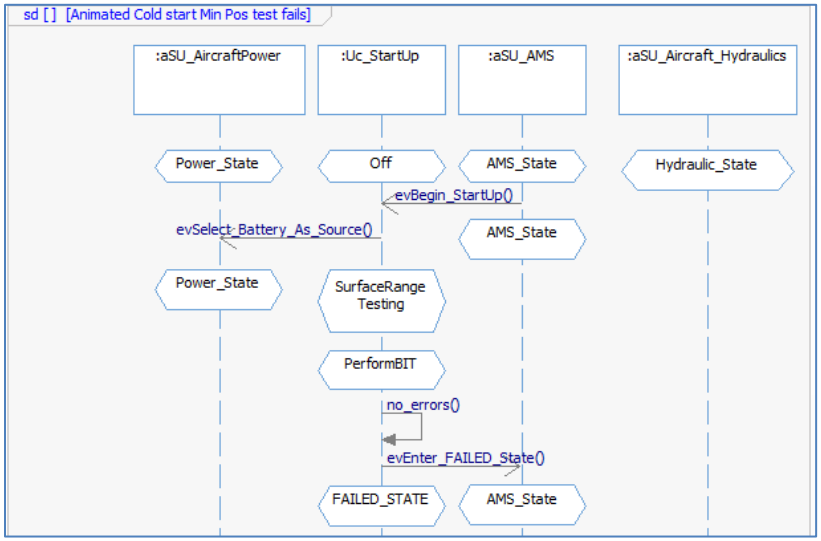


Figure 97: Simulation of error flow

So far, setting values is done by changing the model and recompiling. This is a little tedious. Rhapsody provides a number of ways to change values during run time, including the Webify feature, panel diagrams, or simply adding special events for the purpose of changing values to run specific test cases (something we'll do in Phase 2)[8].

Before we move on, be sure to set the default value of **error_count** back to 0.

Phase 2: Add movement tests

We've had a successful first simulation run, but we didn't model all the requirements. Specifically, the use case block states **SurfaceRangeTesting** and **PerformingBIT** didn't actually do anything. In this phase, we'll add a submachine to the **SurfaceRangeTestings** state to model those requirements.

Open the state machine for the use case block **Uc_StartUp**. Right click on the state **SurfaceRangeTestings** and select *Create Sub-Statechart*. This will create another state machine diagram which is logically a part of its parent. This is a good way to manage large, complex state machines. Select the **SurfaceRangeTestings** state on the newly created diagram and drag a corner to make it large. Everything you do on this diagram must go inside that composite state. We will base this state behavior on the activity diagram specification we made at the start of this use case analysis (Figure 58). It will look a bit different because this is a state machine and not an activity diagram (and we want it to actually execute), but it's lineage should be obvious.

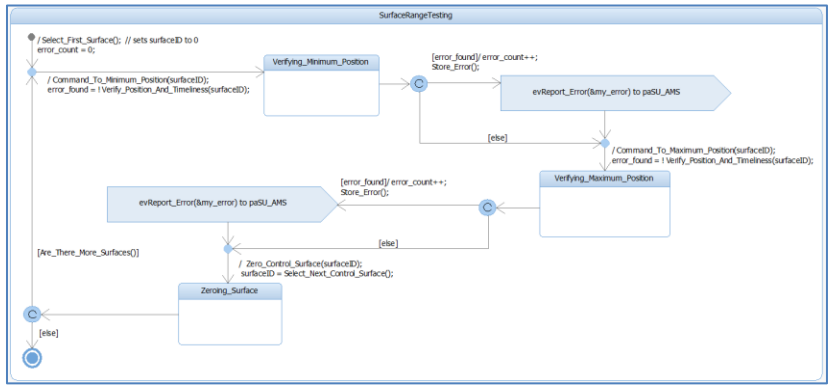
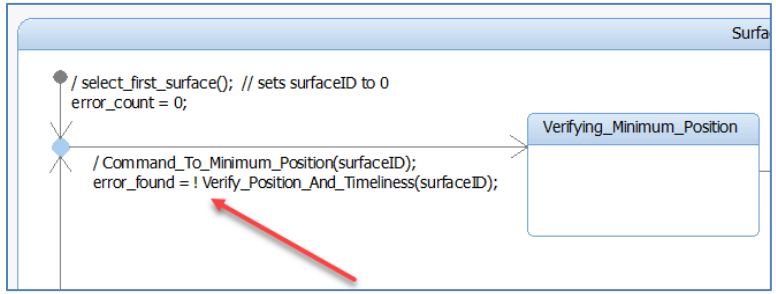


Figure 98: Surface Range Testings Submachine

Note that **error_found** is set to the **NOT** ("**!**") of **Verify_Position_And_Timeliness().**



[8] There are many short but useful videos on YouTube on the use of these features. You might start here to search for video content of interest: https://www.youtube.com/watch?v=zODaYlqL1_A&list=PL1122E405F2CC4EE5

Figure 98 is the state machine equivalent of activity diagram in Figure 58. To do this, we've used eight operations that will need some implementation for the simulation:

- **Select_First_Surface()**
- **Command_To_Minimum_Position(surfaceID);**
- **Verify_Position_And_Timeliness(surfaceID);**
- **Store_Error();**
- **Command_To_Maximum_Position(surfaceID);**
- **Zero_Control_Surface(surfaceID);**
- **Select_Next_Control_Surface();**
- **Are_There_More_Surfaces(surfaceID);**

and three value properties:

- **surfaceID** (of type **int**) (holds the number of the surface of current interest)
- **error_found** (of type **RhpBoolean**) (holds whether errors have been found)
- **pos_ok** (of type **RhpBoolean** and a default value of *TRUE*) (holds whether the surface correctly achieved commanded position)

The operations were added to the use case block **Uc_StartUp** during the creation of the sequence diagrams from the activity diagram; the value properties must be added as new elements. Let's deal with the value properties first. Using the same method we used to add **time_since_last_reset** and **error_count** in Phase 1, add these two new variables, giving **surfaceID** a default value of 0 and **error_found** a default value of **FALSE**.

Adding the operation implementations is similarly easy.

- **Select_First_Surface()**
  set the implementation to
  ```
  surfaceID = 0;
  ```
- **Command_To_Minimum_Position(surfaceID);**
  No implementation needed to support the simulation at this time. Add an argument **sID**, of type int but you needn't do anything with it.
- **Verify_Position_And_Timeliness(surfaceID);**
  This should return a RhpBoolean. Add an argument **sID**, of type *int* but you needn't do anything with it. The implementation can simply be
  ```
  return pos_ok;
  ```
- **Store_Error();**
  No implementation necessary.
- **Command_To_Maximum_Position(surfaceID);**
  No implementation needed to support the simulation at this time. Add an argument **sID**, of type int but you needn't do anything with it.
- **Zero_Control_Surface(surfaceID);**
  No implementation needed to support the simulation at this time. Add an argument **sID**, of type int but you needn't do anything with it.
- **Select_Next_Control_Surface();**
  Set to return an int value and add the following implementation:
  return `surfaceID +1;`
- **Are_There_More_Surfaces();**
  This is a new operation (it was missed in the scenario creation), so it must be added to the use case block. This must return an RhpBoolean. Add the following implementation:
  ```
  return (surface ID == 0);
  ```

Again, we're trying to support the black box view; we don't really care exactly how things happen inside the system, so we don't have to actually move surfaces around and check them. We just have to simulate what that looks like from an external perspective. That's what this simulation does.

This is set up to run the test for only the first **surfaceID**. If you want to run multiple surfaces, then change the implementation of **Are_There_More_Surfaces()** to return *FALSE* as soon as you've done the desired number of surfaces. In the implementation provided, it only returns *TRUE* if the **surfaceID** is zero. As soon as it is augmented by the operation **Select_Next_Control_Surface(),** then it will return *FALSE*.

The more interesting part is we have to add values to send to the **aSU_AMS** actor block via the **evReport_Event**. You may remember that this event takes an argument err of type **Error_Report** (see Figure 80).

Let's add another value property named **my_error** of type **Error_Report** to the use case block **Uc_StartUp**. We will update the specific fields of this attribute when we send the error report to the **aSU_AMS**. In the *Send Actions* of Figure 98, add **&my_error** to the parameter list[9].

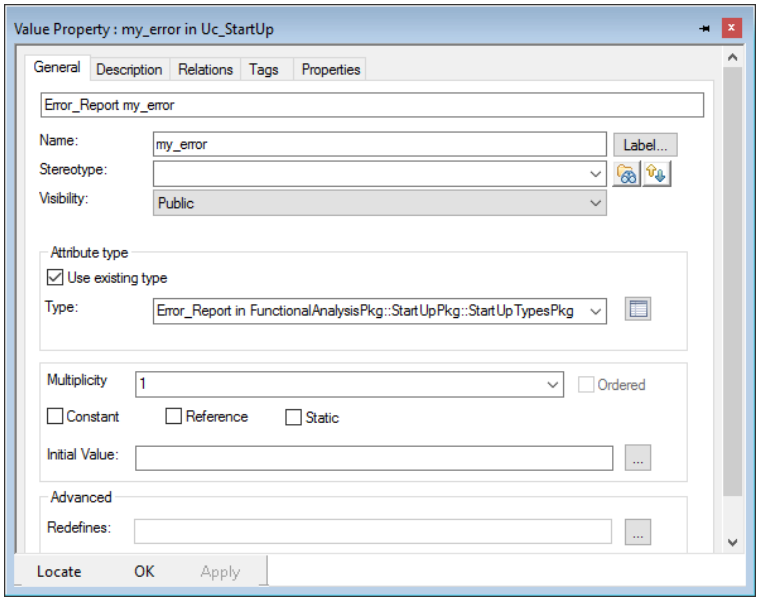


Figure 99: Defining attribute my_error

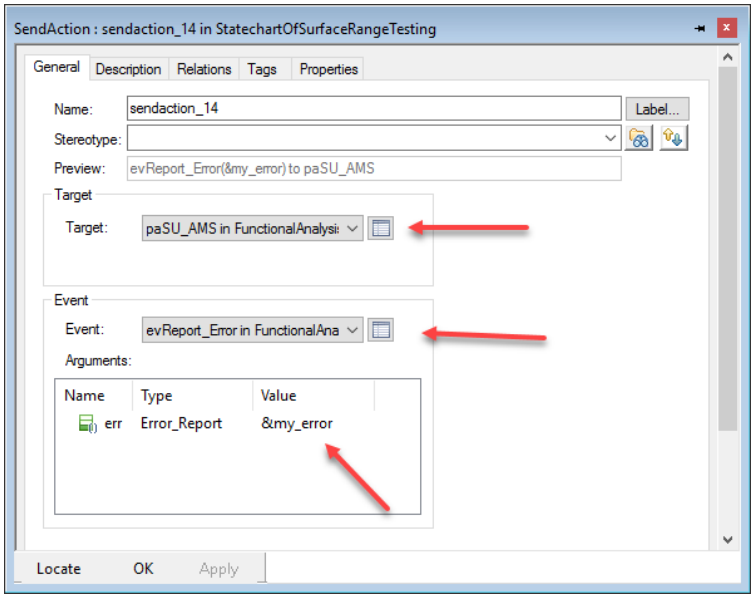


Figure 100: Adding parameter my_error to Send Action

This results in an updated state machine for the use case block. Note that not only do the send actions have parameters for the event being sent, we've also added actions to set up values to identify the errors detected.

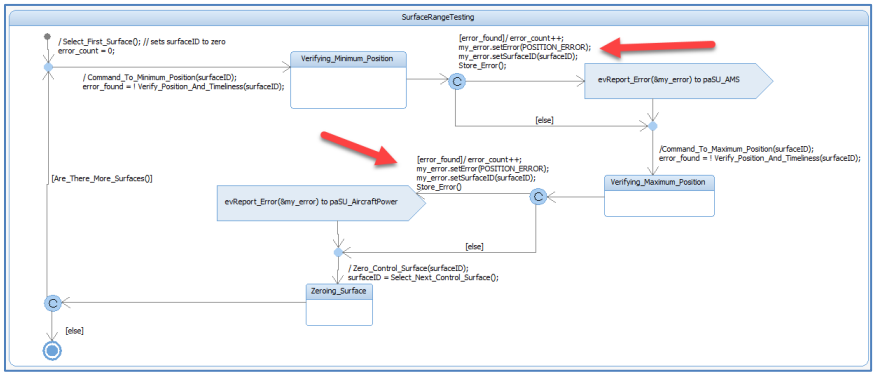


Figure 101: Uc_StartUp state machine updated with event parameters

The next figure shows a close up of the event paramers in use:

[9] The **&** operator is interpreted as "the address of the value property named **my_error**." See Section 12 for details.

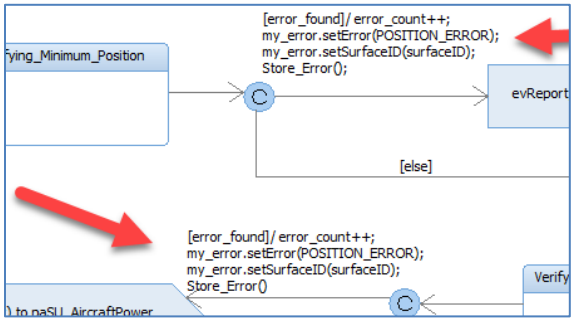


Figure 102: setting up my_error values

The only remaining thing to do before we can run this is to update the **aSU_AMS** state machine to receive the **evReport_Error** event.

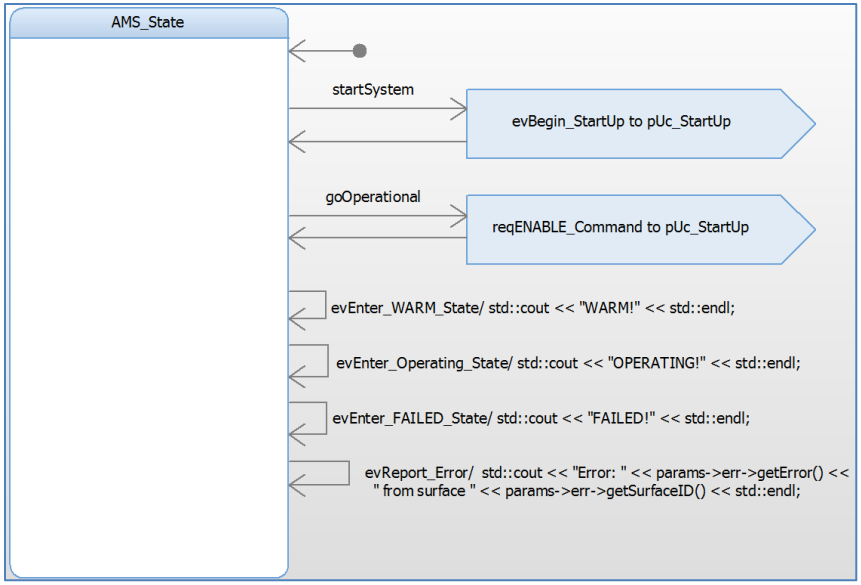


Figure 103: Updated state machine of aSU_AMS

A quick note: it sometimes happens that Rhapsody doesn't quite get the generated dependencies correct. If Figure 101 compile fails with errors accessing the **err** parameter of the **evError_Report**, you might need to go into the browser and add a *«Usage»* dependency from the actor block **aSU_AMS** to the **Error_Report** block in the **StartUpTypes** package.

Let's now run two simulations. The first will find no errors. That should compile and run as-is. The second will require a small tweak.

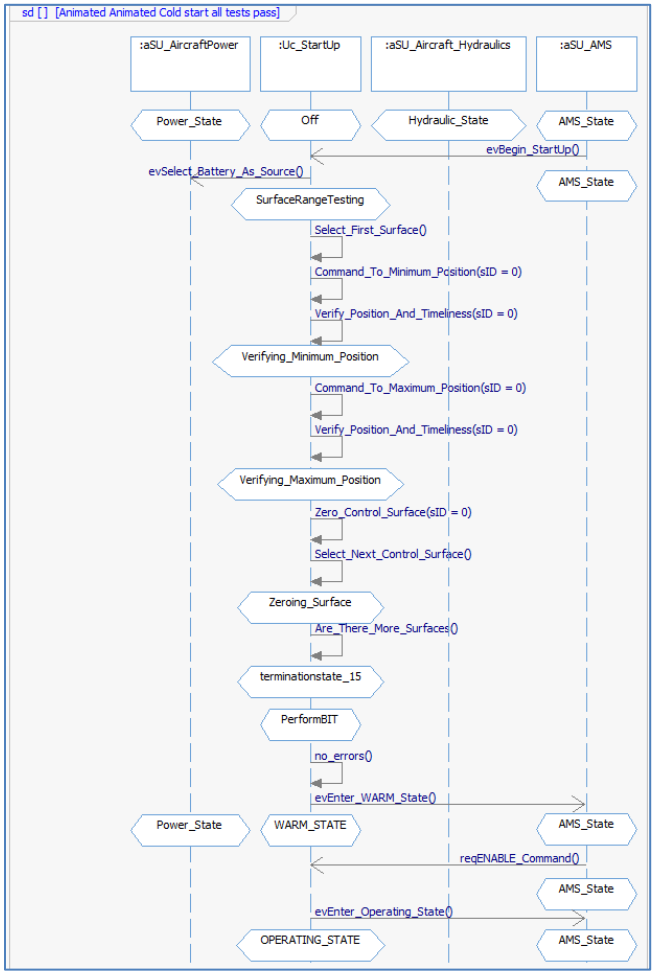
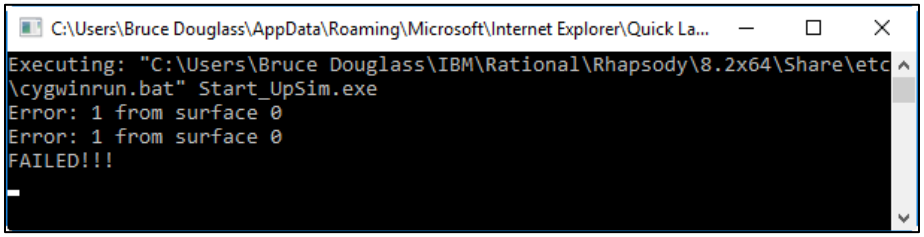


Figure 104: Start Up with surface range tests passed (Phase 2)

Now, let's do the same simulation with the surface ranging tests failed. Stop the execution and edit the **pos_ok** value property of the **Uc_StartUp** block so that its default value is *FALSE*. Now, recompile and run.

The standard output window should look like this:



The scenario result should indicate the system ending in the **FAILED** state.

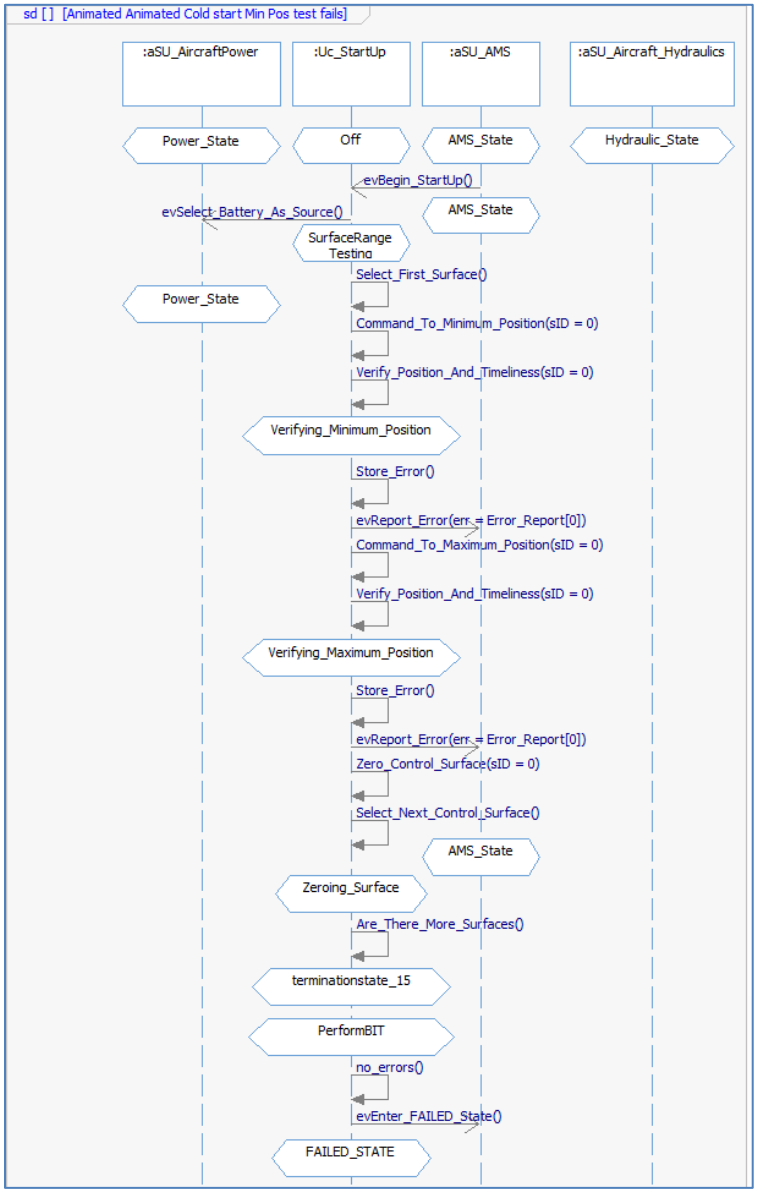


Figure 105: Start Up with surface range tests failed (Phase 2)

Don't forget to change the value of **posOk** value property back to its default value of **TRUE**.

Phase 3: Add power, hydraulic, and SW integrity tests

This nanocycle iteration of the analysis of the **Start Up** use case adds the remaining requirements into the mix.

First, let's add the state behavior for the **Uc_StartUp** block for these tests. Open the use case block main state machine diagram, right click on the **PerformingBIT** state and select *Create Sub-Statechart*.

Just as we added details for the surface ranging tests in the submachine for **SurfaceRangingTesting** state in Phase 2, we'll add the tests for power, hydraulics, and software integrity into the submachine for the **PerformingBIT** state.

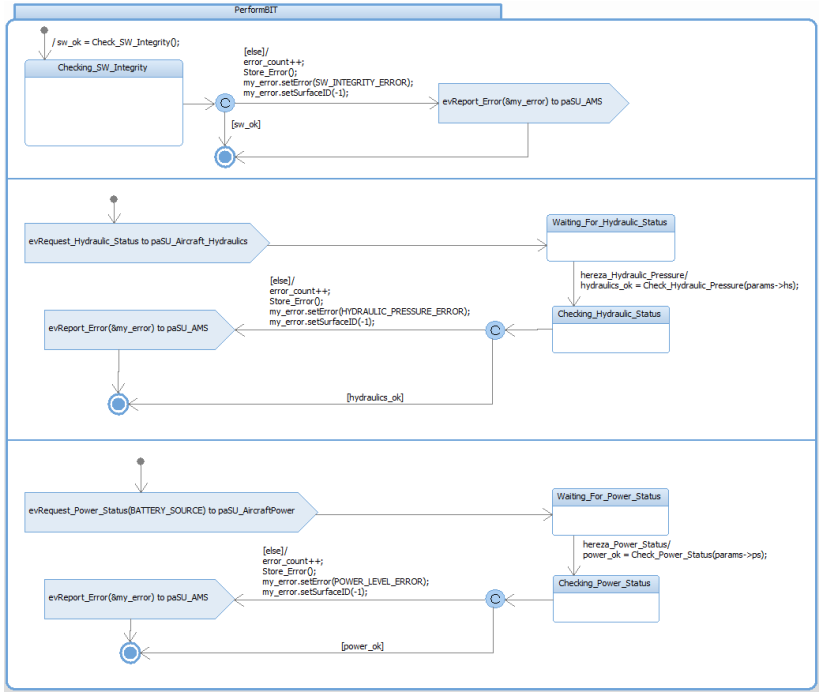


Figure 106: Submachine for PerformingBIT state

Be sure to include the paramers for the **Check_Power_Status** and **Check_Hydraulic_Status** operations, as shown in Figure 106.

In the process of elaborating this state machine, we've added three Boolean variables to the **Uc_StartUp** block (the last one will be used shortly):

- **sw_ok** (default *TRUE*)
- **hydraulics_ok** (default *TRUE*)
- **power_ok** (default *TRUE*)
- **sw_fault** (default *FALSE*)

Go ahead and add them to the **Uc_StartUp** use case block with the specified default values.

The state machine uses another three operations to the block as well:

- **Check_SW_Integrity()**
- **Check_Hydraulic_Pressure(h: HydraulicStatus);**
- **Check_Power_Status(p: PowerStatus)**

Be aware that the SE Toolkit has previously added these functions when we parsed the activity diagram to create the scenarios. Nevertheless, we must still add parameters and implementation.

In the browser, double click on the **Check_SW_Integrity** operation to open its *Features* dialog. In the *General* pane, change the return type from **void** to **RhpBoolean**. In the implementation pane, add the implementation

```
return ! sw_fault;
```

(be sure use to include the **NOT** ("!").).

Then add **sw_fault** as a value property (or attribute) of type **RhpBoolean** to the use case block **Uc_StartUp**, and give it a default of **FALSE**. Having this as an internal variable allows us an easy way to simulate cases where this test passes or fails.

Next, update the **Check_Hydraulic_Pressure** operation. In the *General* tab, make sure it returns an **RhpBoolean** value. In the *Arguments* pane, add an argument **h** of type **Hydraulic_Status**. Note that Rhapsody will automatically pass a pointer to the type[10]. Now add the following to the *Implementation* tab:

```
return ! h->getHas_faults();
```

Similarly, update the **Check_Power_Status** function to return an **RhpBoolean**, add an argument **p** of type **Power_Status**, and add the implementation:

```
return ! p->getHas_faults();
```

Now let's update the actor blocks.

The **aSU_Aircraft_Power** and **aSU_Aircraft_Hydraulics** blocks are really pretty simple. Add the follow state machines to these blocks (don't forget they are located in the nested package **StartUpActorPkg**.

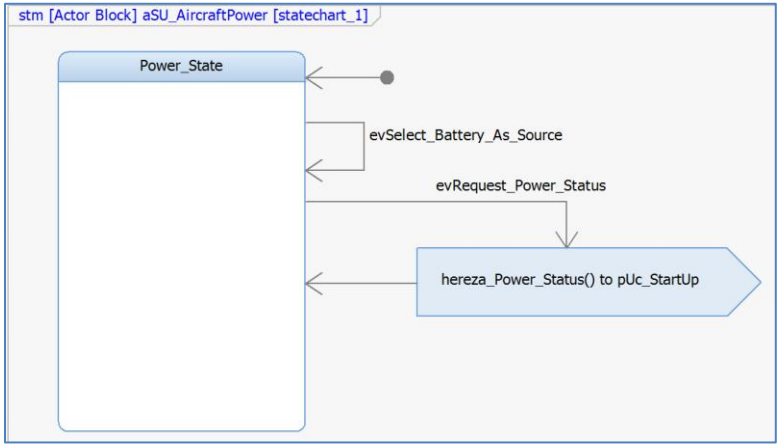


Figure 107: aSE_Aircraft_Power state machine

[10] Again, see Section 12 for more details on parameter passing.

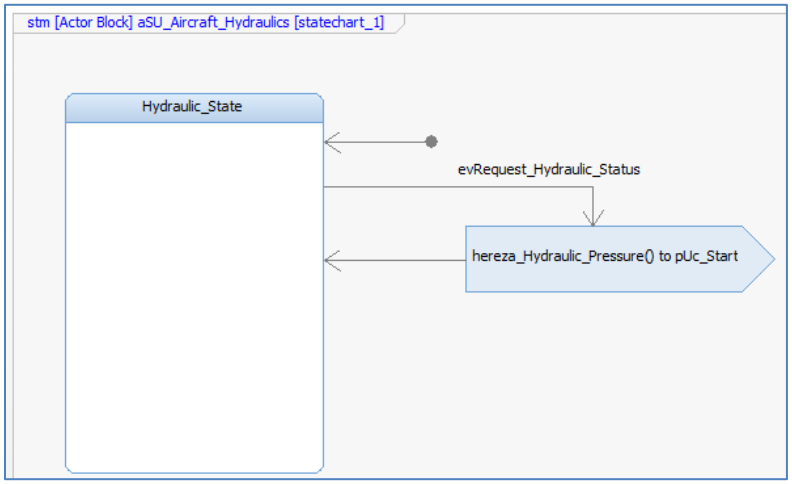


Figure 108: aSU_Aircraft_Hydraulics state machine

Ok, hold on. It's not *quite* that simple. These state machines send events **herezaPowerStatus**[11] and **herezaHydraulicStatus**, both of which pass data. We'll need to create local attributes of type **Power_Status** and **Hydraulic_Status** and pass the values along with the events.

Right click on the actor block **aSU_Aircraft_Power** and select *Add New > Value Property*. Name this attribute **p_status**. Double click on this to open the *Features* dialog. In the General pane of the dialog, choose the *<<Select>>* option in the drop down list and navigate to the **StartUpTypesPkg** to select the **Power_Status** block as the type.

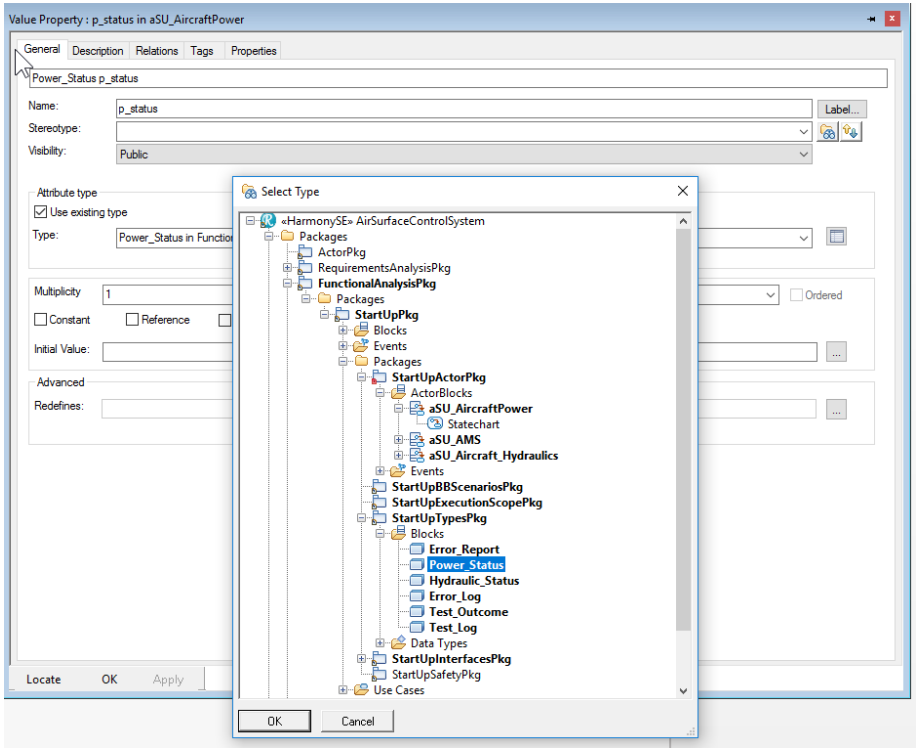


Figure 109: Selecting a type from the StartUpTypesPkg

At run-time, we want to be able to provide status that is either good or bad. We'll do this by elaborating the actor block state machine. This means that we can, at run-time, easily send either value during a simulation.

[11] "hereza" as in "here is a … "

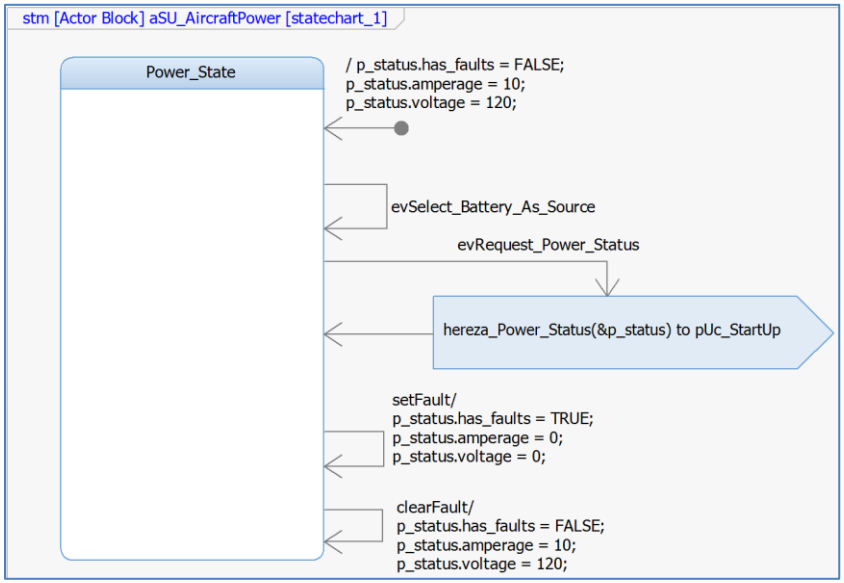


Figure 110: Setting power status values during simulation

Figure 110 shows how we do this. We've added **setFault** and **clearFault** events to set appropriate values for the **p_status** attribute. And we updated the *Send action* for **herezaPower_Status** to pass the value **"&p_status**" (read as "the address of the value named **p_status**").

We must now do the equivalent thing for the hydraulic status. This will require similar modifications to the state machine for the block **aSU_Aircraft_Hydraulics**, although we will add a value property named **h_Status** (of type **Hydraulic_Status**) to the actor block.

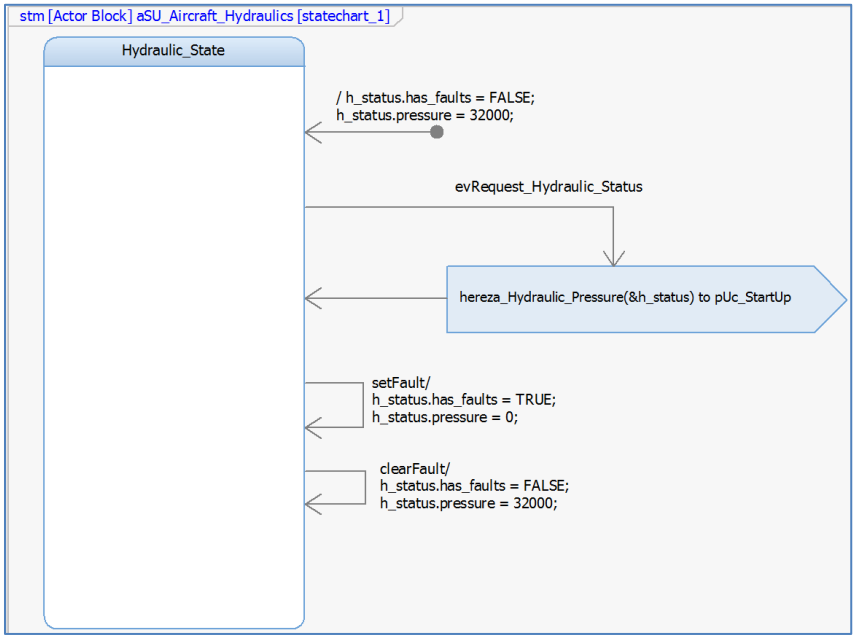


Figure 111: Setting hydraulic status values during simulation

Let's now run the case in which all tests pass[12].

[12] In the actor blocks, we could use the automatic accessor and mutator operations but we're now directly accessing attributes. If you haven't already done this, you can change the Rhapsody settings to allow this by right-clicking on the project in the browser going to the *Properties* Pane, select *View All*, and going to the topic *CG_CPP > Attribute > Visibility*. Here you have a drop down list. The default visibility is set to *protected*, but you can select *fromAttribute*.

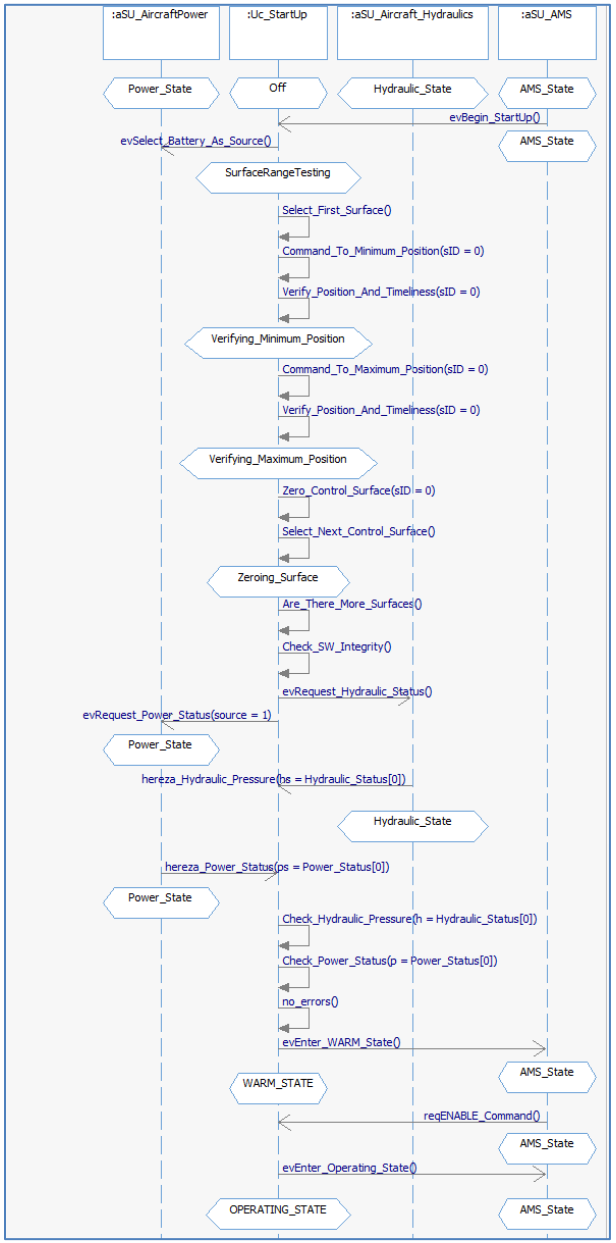


Figure 112: All tests pass (Phase 3)

Figure 112 shows the successful start up of the system with all tests modeled and passing. I removed some of the state condition marks on the generated sequence diagram to shorten it a bit.

Let's together do one more scenario. In this scenario, we'll start the system running, but before we signal the system to proceed (by generating the **startSystem** event), we'll set the hydraulic system to have an error. We ll do this by sending the event setFault to the aSE_Aircraft_Hydraulics actor block after we start the system. We want to ensure that the system properly detects the error and reports it to the **AMS**.

Run the simulation by clicking on the **Go** button on the animation toolbar. If you don't have an animated sequence diagram open to capture the flow, open one with the *Tools > Animated Sequence Diagram* menu option, selecting one of the sequence diagrams in the **FunctionalAnalysisPkg > StartUpPkg > StartUpBBScenariosPkg** package.

Right click on the :**aSU_Aircraft_Hydraulics** lifeline on the animated sequence diagram and select *Generate* Event. In the drop down box, select the **setFault** event and click on *Generate*. Then, in the *Generate Event* dialog box, select the **aSU_AMS** object and generate the event **startSystem**. It should generate something like Figure 113. Note that I added the *System Border* lifeline so that the scenario shows the user-entered events. The resulting scenario is shown in Figure 114. This figure has also been edited a bit to remove some condition marks and comments were added.

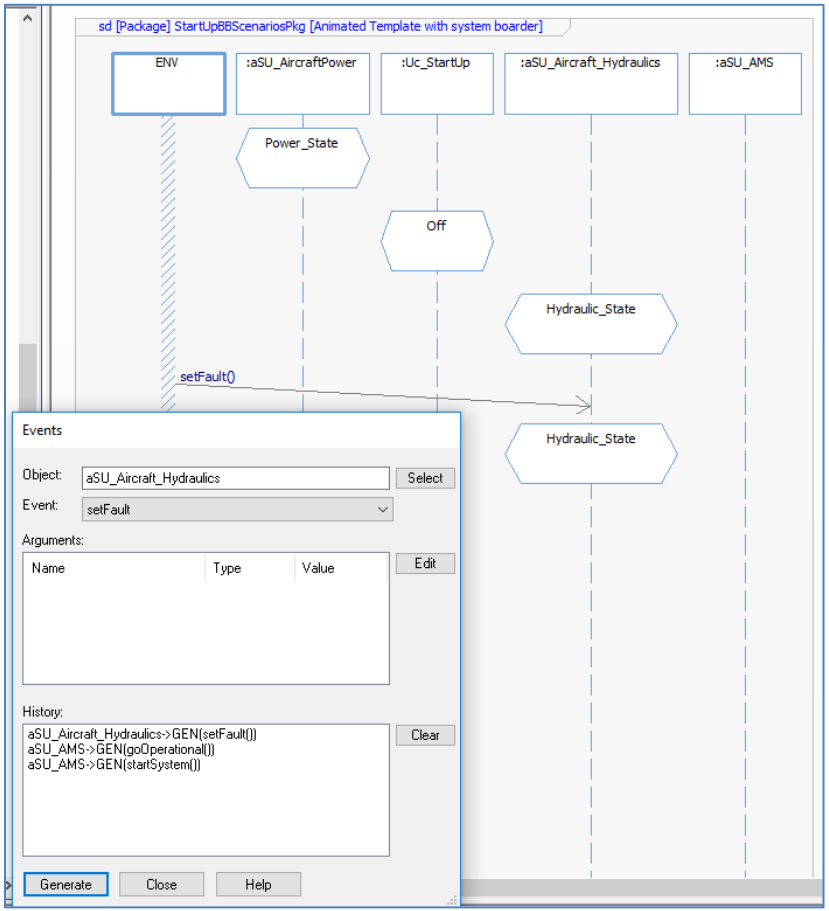


Figure 113: Entering the setFaults event to simulate error

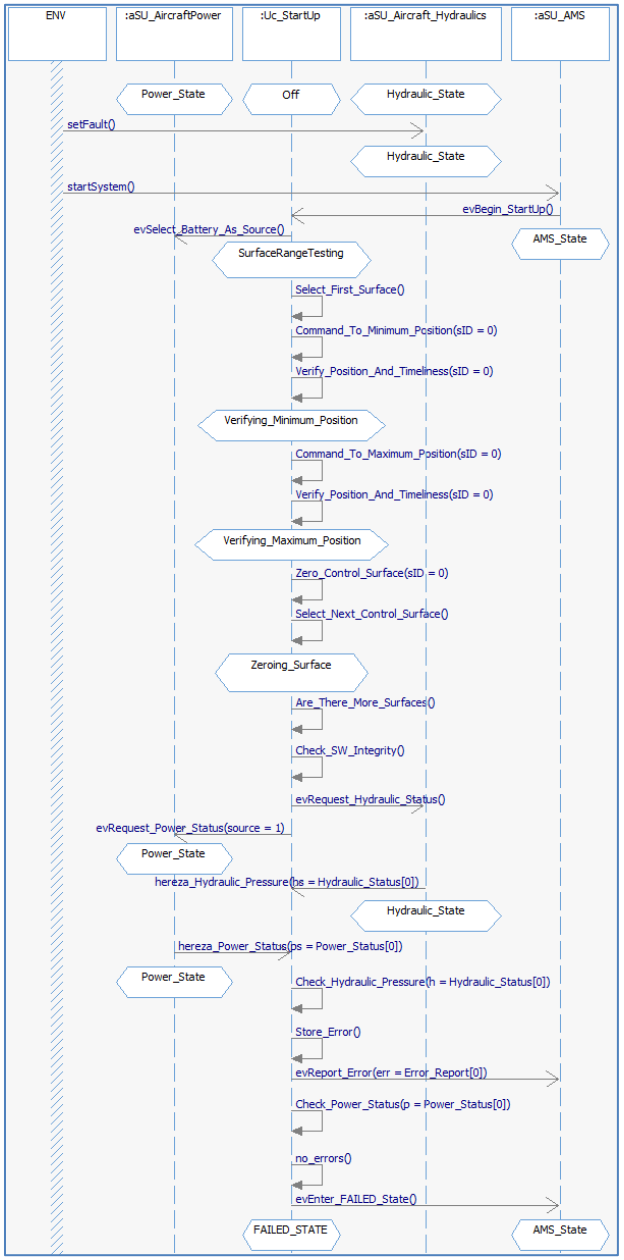


Figure 114: Cold Start Up with hydraulic test failure

Now, on your own, generate the sequence diagrams for the case where the SW integrity fails and one for when the Power System fails.

We have now completed the black box analysis of the **Start up** use case. We did this using an incremental approach following the system function-based approach outlined in Figure 53. We have traceability from the use case to the requirements, but we have not added more detailed traces from the use case block value properties, event receptions, and operations to the requirements.

> In a fully formed systems engineering model, all normative model elements have descriptions and all elements have trace relations. If a value property, an event reception, and two operations all trace to a single requirement, then you should add these relations.

Let's now move on to the second use case we will study.

7.4 Analyze the Control Air Surfaces Use Case

This use case is considerably more complex than the previous use case and so its analysis will require correspondingly more detail.

This use case will be analyzed using the interaction-based (or scenario-based) approach from Figure 4. A detailed look at the workflow is shown in Figure 115.

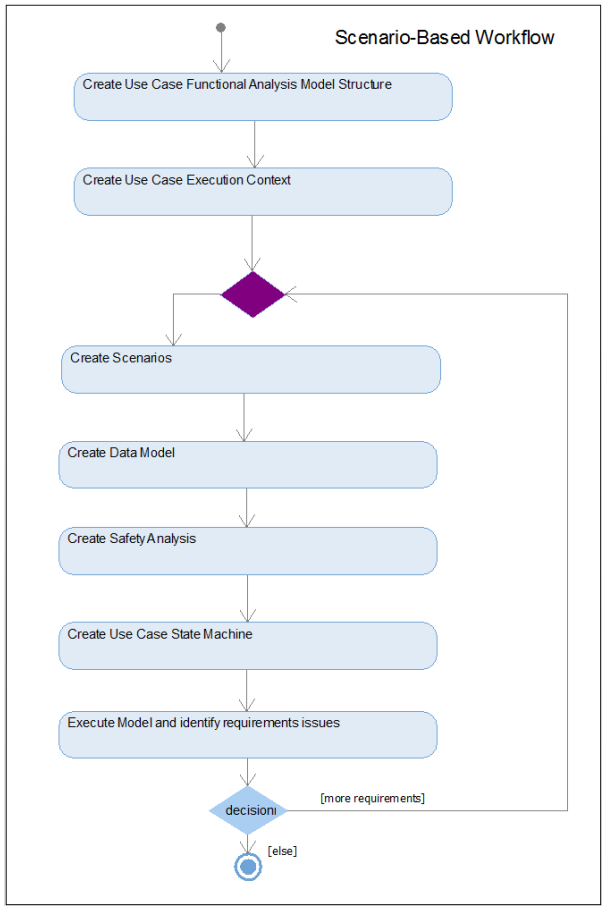


Figure 115: Detailed Work Flow for Scenario-Based Use Case Analysis

7.4.1 Create Use Case Functional Analysis Model Structure

The SE-Toolkit provides a handy tool for the creation of the internal package structure for the **Control Air Surfaces** use case analysis within the **FunctionalAnalysisPkg** package. Right-click on the use case in the browser and select *SE-Toolkit > Create System Model from Use Case* (Figure 116).

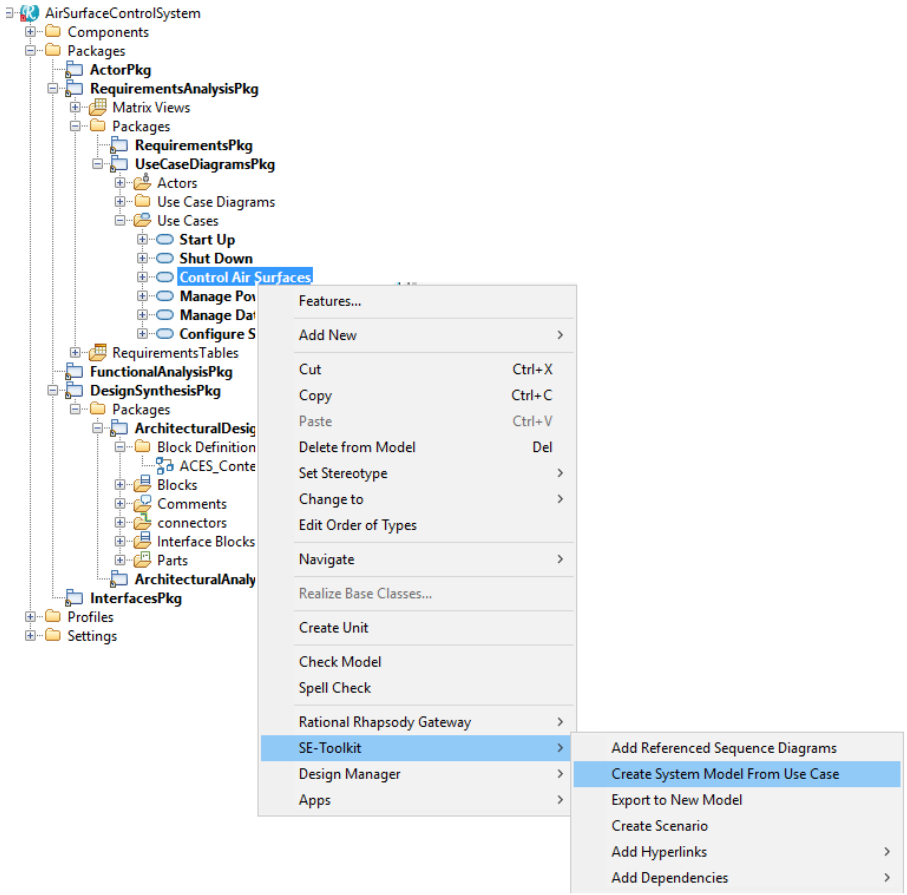


Figure 116: Using the Create System Model from Use Case tool

Similar to our use of this tool for the **Start Up** use case, the tool creates a package called **ControlAirSurfacesPkg** and then populates it with the appropriate blocks for the use case and actors, creates the appropriate links and even creates a new internal block diagram (IBD) showing the use case execution context. The **ControlAirSurfaceExecutionScopePkg** also contains a new component named **Control_Air_SurfacesSim** for building the executable model (to come later). The fully elaborated package structure for this functional analysis package is shown in Figure 117.

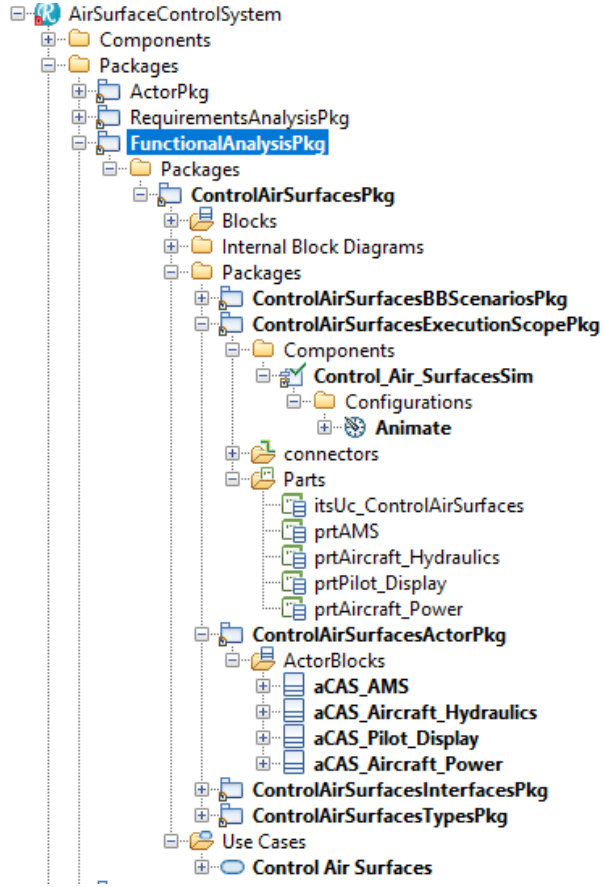


Figure 117: ControlAirSurfacePkg structure

The IBD created by the toolkit and subsequently beautified manually is shown in Figure 118.

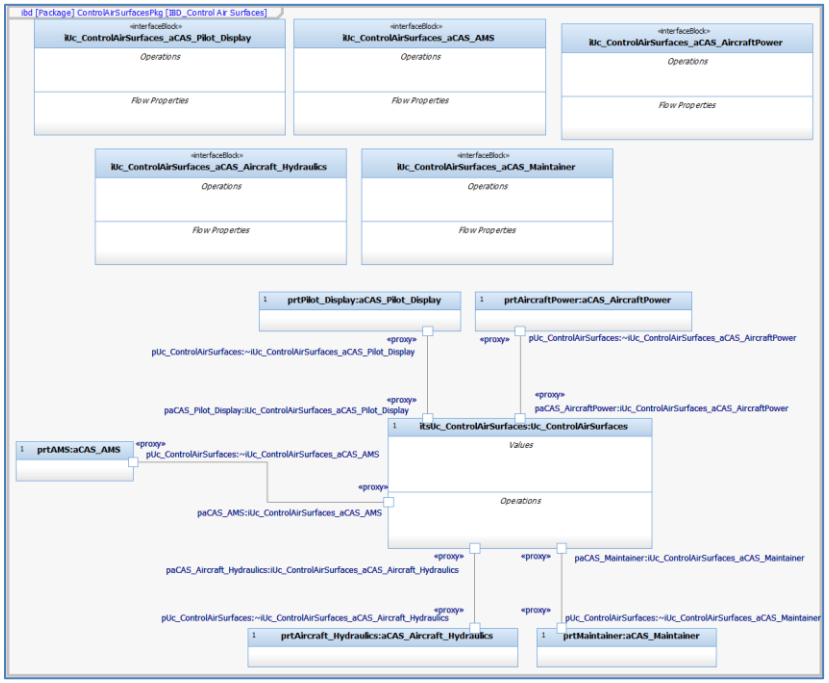


Figure 118: Beautified Control Air Surfaces Use Case Execution Context IBD

7.4.2 Create Scenarios

Since we are using the Scenario-based use case analysis strategy from Figure 4, the next thing to do is start creating scenarios of interest. This strategy is appropriate when

- working with non-technical stakeholders to identify or explore the requirements
- the use case is primary interaction-based, that is, the focus of the use case is on the interaction of the system and its actors rather than on internal functionality

The SE Toolkit provides a tool for creating new scenarios that share a common lifeline structure. If you used the *Create System Model From Use Case* tool previously, then the SE Toolkit already created an activity view under the use case (which it moved to its own package in the **FunctionalAnalysisPkg** package). Right-click the **Control Air SurfacesBlackBoxView** activity diagram under the use case *Activity Views*, and select *SE-Toolkit > Create Scenario*. This diagram should look something like Figure 119.

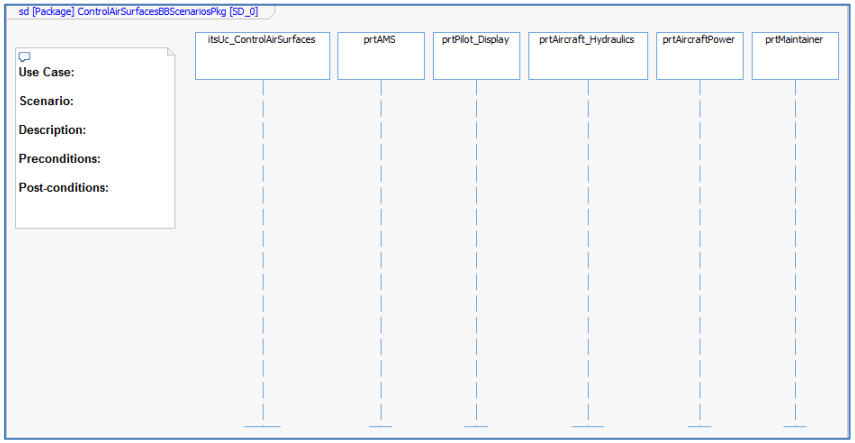


Figure 119: Created Scenario Diagram

Note that it has a template comment for you to elaborate information to help readers understand the scenario you are about to enter.

Creating Good Scenarios: Best Practices

Almost everybody asks at this point "How do I know when I am done?" After all, there are an infinite set of possible scenarios if you consider all variations of sequence, timing, values, responses, and iterations. There are several different answers to this question which are essentially equivalent:

- When every requirement allocated to the use case is expressed in at least one scenario
- When all "interestingly different" flows have been captured
- When every path and action on defining activity diagrams or state machines are represented
- When all the normal path, or "sunny day" scenarios are captured AND all the exceptional, or "rainy day" scenarios are captured

We call such as set of scenarios, the *minimal spanning set,* as it fully represents all the requirements of the use case.

Recommendations

- The point of scenarios is not to show internal functionality as much as to show the interaction of the elements of interest. Thus, messages between the use case and the actors will be modeled as events (possibly carrying data) or flows. "Messages to self" in the use case lifeline represent the execution of *system functions*.
- For sequences in which the flow is identical but the actual values passed are different, it is enough to show a single scenario, but you can add comments or constraints that show the range of values or events that can participate. If the flows are different because the values differ, then this warrants a new scenario.
- Start with the sunny day scenarios and once normal functionality is established and understood, add rainy day scenarios.
- As you create the scenarios, you identify requirements that are missing, incorrect, incomplete, or inaccurate. At that point, add the new requirements into your textual requirements, allocate them to the use case, and express them in one or more scenarios.
- Use events for discrete messages between the actors and the use case and operations for system functions
- Use flows for continuous values.
    - Most commonly, flows are only shown in the sequences at the point at which the value changes
    - For continuous flows, stereotype the flow as «continuous» possibly within a continuous interaction fragment. Note: you may have to right-click and check *Show Stereotype* on the *Display Options* of the flows to see the stereotype after you've added it.

- Most sequence diagram operators – such as loop, optional, and alternative – are just a means by which multiple scenarios can be represented on a single diagram. We recommend not nesting interaction operators more than three levels deep or you risk creating unreadable sequence diagrams.

Creating the First Sunny Day Scenario

Open the diagram you just created (if it's not now open) and add the description text to the comment to the upper left hand corner:

```
Use Case: Control Air Surfaces
Scenario: Scenario 1
Description:
Normal operation, no faults.

Preconditions:
System has passed self-tests without error. System
is in an Inactive condition (WARM state).

Post-conditions:
After cooling, the system goes to Inactive
condition.
```

Rename the diagram **Control Air Surface Scenario 1**. Draw the flows as shown in Figure 120

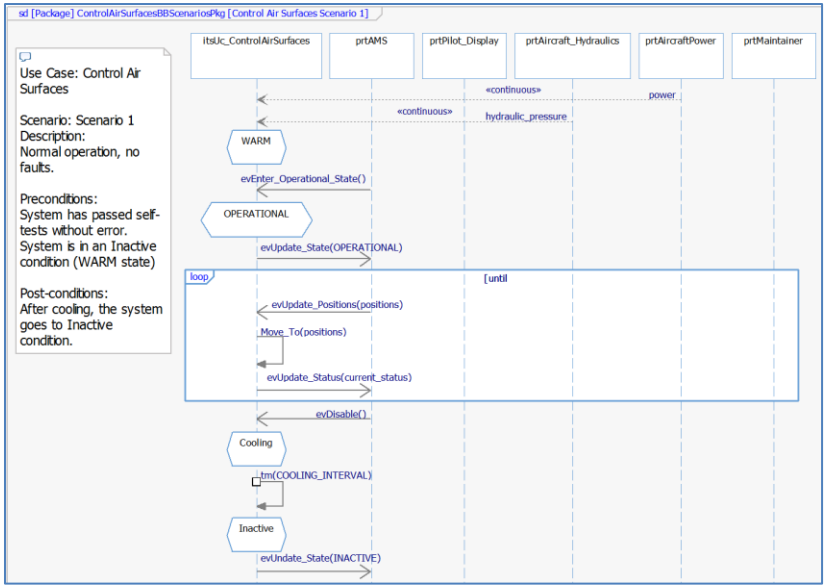


Figure 120: Control Air Surfaces Scenario 1

The scenario shows the recommended descriptive comment in the upper left hand corner[13] and the flows. The scenario starts out showing that the aircraft power and hydraulics continuously providing energy and pressure. Since they don't vary in the scenario, showing these flow at the top of the sequence is fine. Note that this is not the «continuous» provided by the SysML profile (which only applies to *ObjectNode*, *Pin*, *State*, and *Transition*); rather this is the one provided in the Harmony SE Profile.

> Note that I often define stereotypes is likely to be used multiple places in the model, so I add a package at the system level called **CommonPkg** and added those stereotype there. In this case, the stereotype is provided by the Harmony SE Profile.

When the **AMS** decides some or all of the control surface positions should change, it sends an **evUpdate_Positions(positions)** message. The use case then executes the **Move_To(positions)** system function and then responds with its current status. The data elements **positions** and **current_status** are identified and used here but are not detailed. We will detail them in the logical data and flow schema.

These activities continue until the **AMS** disables the movement. This is shown with the *loop* interaction operator. At that point, the use case enters a **Cooling** period. After the cooling period is over, the system becomes **Inactive** and notifies the **AMS**. The **Cooling** period is necessary to support rapid restarts by the pilot (via the **AMS**), should that become necessary.

Creating the Second Sunny Day Scenario

The next sunny day scenario elaborates the first scenario. The system goes operational, as before, but then is re-enabled during the cooling period.

To make this scenario shorter, we will abstract the parallel region of normal operation into a separate sequence diagram and then reference on the main scenario. To do this, complete the following steps:

- Create a new sequence diagram as before and name it **Control Air Surface Normal Operation Fragment**.
- Add descriptive text to the comment:

```
Scenario: Fragment of normal
operations for Control Air Surface use
case

Description:
Just shows normal flow while moving
control surfaces

Preconditions:
System has entered normal control of
air surfaces

Post-condition:
System is in the process of
terminating normal control behavior

Invariants:
No errors are found
```

You should now have a diagram named **Control Air Surface Normal Operation Fragment** that looks like Figure 121:

[13] Some people prefer to add this information in the description field of the features dialog of the diagram. I prefer this because of its visibility. Either is ok.

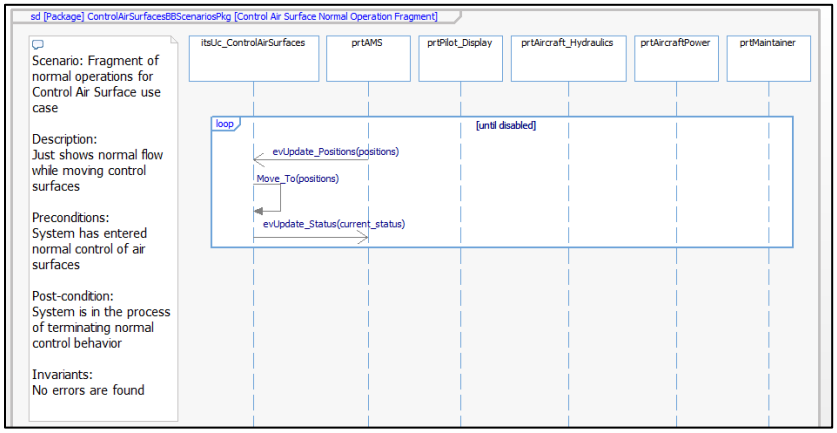


Figure 121: Referenced Interaction Fragment

Now we can use this to build the second sunny day scenario.

- Create a new sequence diagram and name it **Control Air Surface UC Scenario 2**
- Fill out the sequence diagram as shown.
  - To add the reference to the normal operation fragment, add an *Interaction Occurrence* from the toolbar. Once placed, double click and select **Control Air Surface Normal Operation Fragment** from the list of possible sequence diagrams
  - Notice the use of the **CanTm()** (cancelled timeout) message following the first **Cooling** condition. That's to indicate the system timing was interrupted by an arriving **eventer_Operational_State** message.

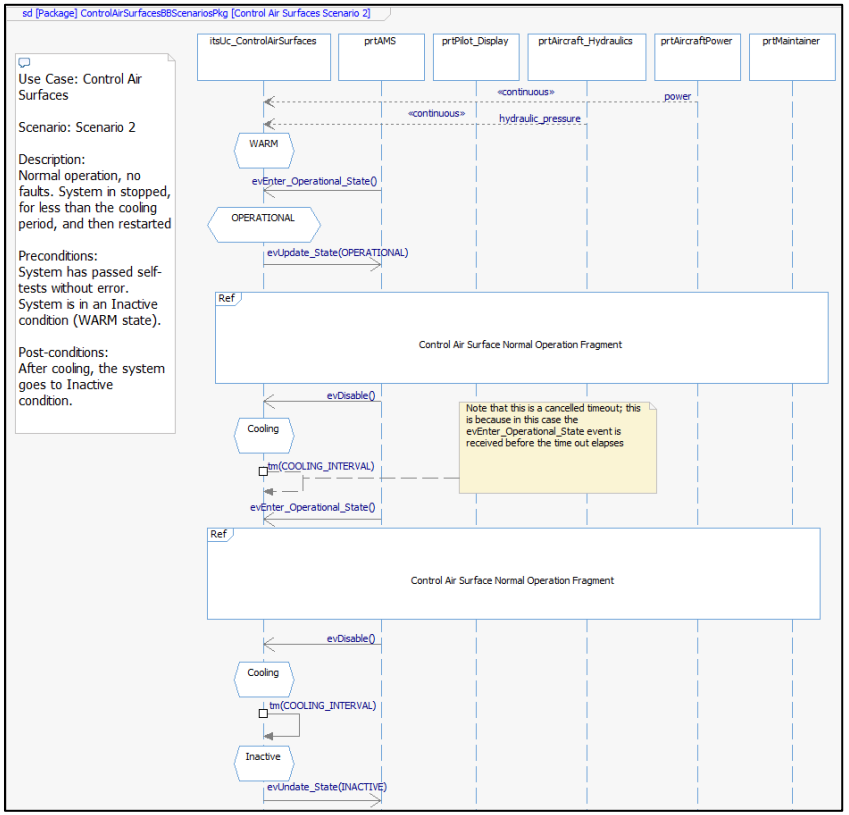


Figure 122: Control Air Surfaces UC Scenario 2

For the 3rd scenario, let's try to start the use case up after the system has failed its power on self test (as detailed in the **Start Up** use case analysis). Repeat the scenario creation procedure as before to create the **Control Air Surface UC Scenario 3** sequence diagram.

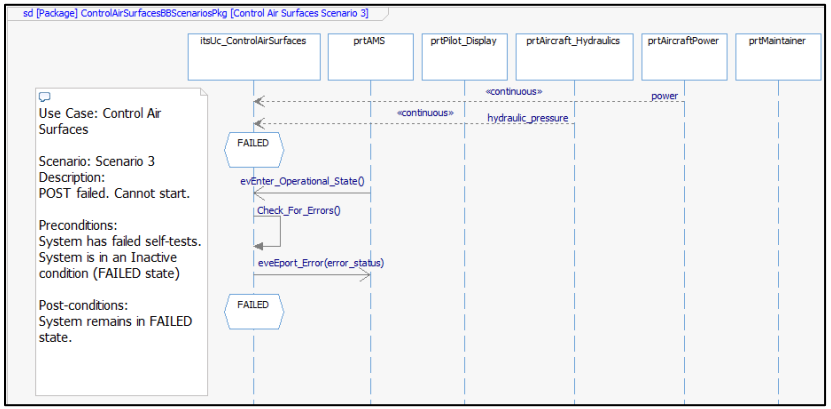


Figure 123: Control Air Surface Scenario 3

For our final scenario in this example, let's consider what should happen if an accuracy or position error occurs in the critical flight control surfaces when commanded to a new position. Both timing and accuracy errors are treated the same, so it is enough to show a single scenario for both with a constraint identifying the conditions that are consistent with the scenario. In addition, sufficiently severe errors in power or hydraulic pressure can also result in the system shutting down.

To show this scenario, we've used an interaction fragment called **Control Air Surface Unflyable Operation Fragment**. First, here's the main scenario.

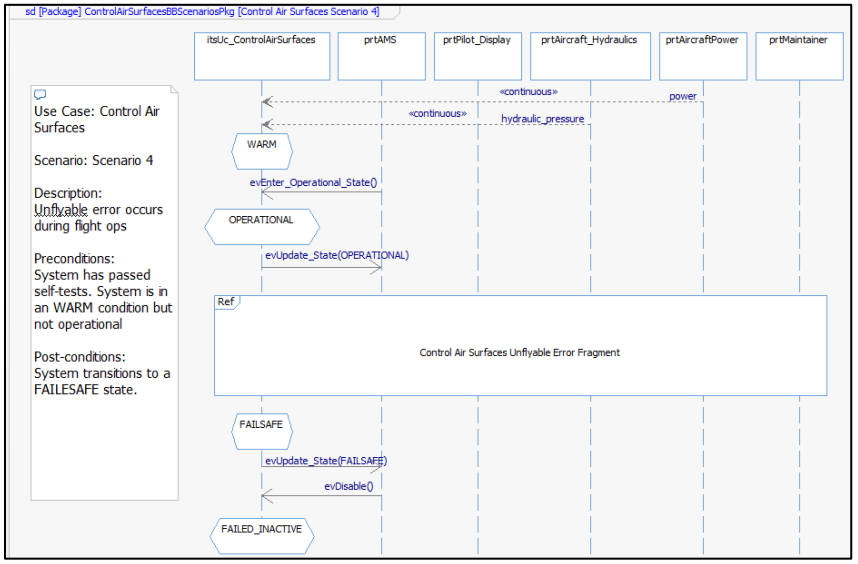


Figure 124: Control Air Surfaces Scenario 4 main

The detail for the actual error handling and detection is shown in Figure 125.

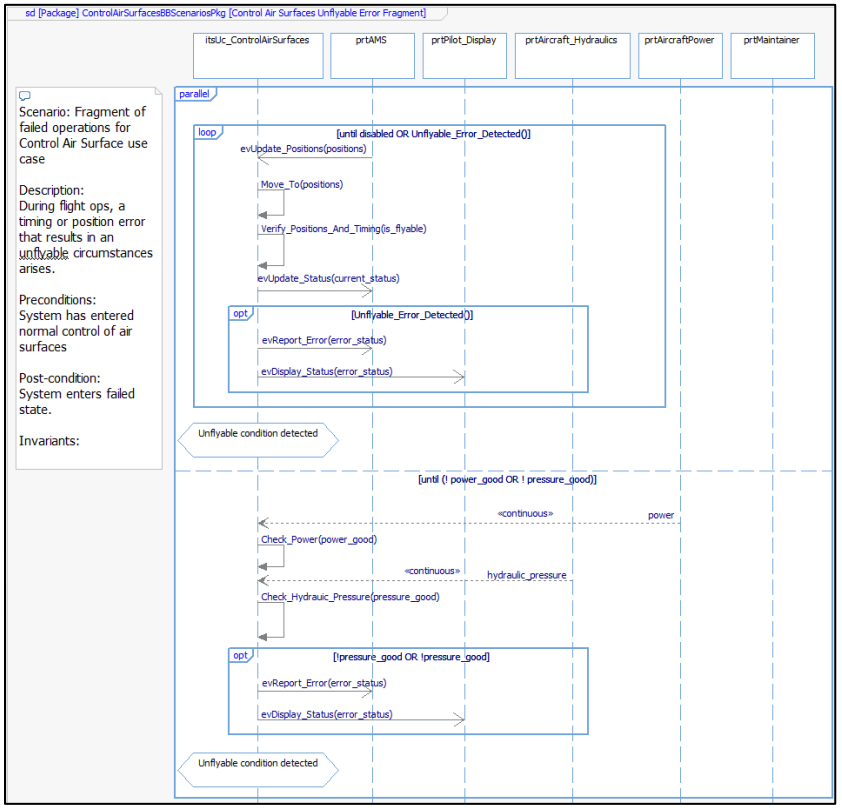


Figure 125: Unflyable Interaction Fragment

At this point, you may autorealize the messages on the created sequence diagrams and run the *SE Toolkit > Ports and Interfaces > Create Ports and Interfaces* tool to add the event receptions and operations to the actor and use case blocks. This is the outcome, as shown in the browser:

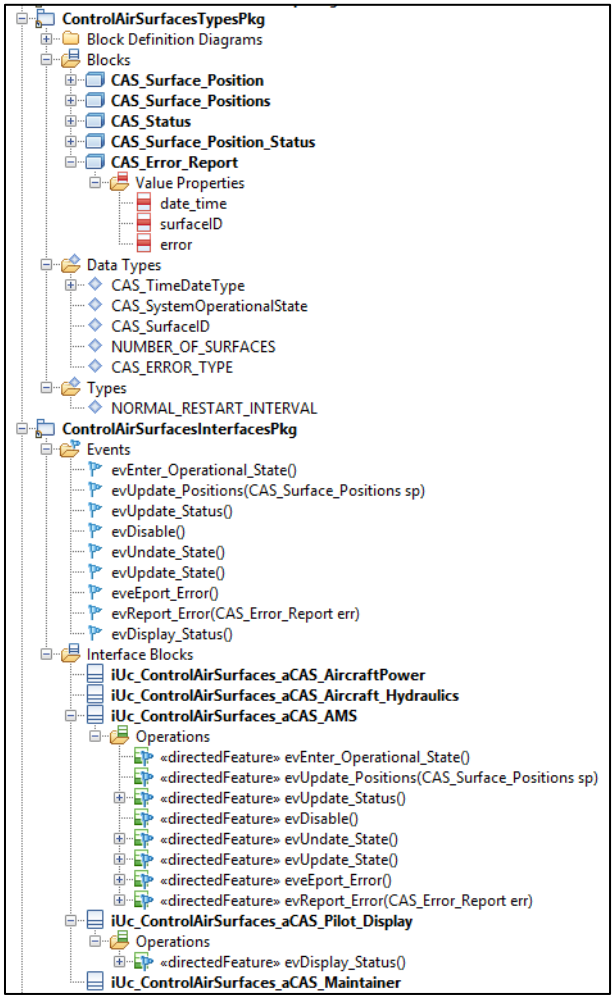


Figure 126: Interface details added for Control Air Surfaces Use Case

7.4.3 Creating the Logical Data and Flow Schema

There are many more scenarios that could be added. We'll stop here in interest of keeping the Deskbook short. Now we can proceed to the next step in the process, which will be to create the logical data and flow schema (which we will just call the "data schema" for short) for the values and flows we've identified. The data schema will be placed in the

FunctionalAnalysisPkg > ControlAirSurfacesPkg > ControlAirSurfacesTypesPkg package. This package will contain block definition diagrams (BDDs) showing the schema as well as the types and their relations. Note that we are not identifying types *internal* to the system at this point, but rather focused on the types that must be exchanged via the interfaces defined for the use case block.

What data and flows can you identify from the scenarios we've identified? The most obvious one is the data to set the positions of the control surfaces that comes from the **AMS** (or in this case, the **aCAS_AMS** actor block) used in the message **evUpdate_Positions()** and **Move_To()** system function in Figure 120 and Figure 121. Also note the response of the **evUpdate_Status()** message in response to that. What information should that pass back to the **AMS**?

In discussion with the **AMS** stakeholders, we discover that they would like to set all positions on every **evUpdate_Positions()** message, even if the position hasn't changed. Further, they stated that they would like the following information back in the **evUpdate_Status()** message:

- For each control surface, its current commanded and measured position, time when the measurement was taken, and the time necessary to achieve that position from receipt of the movement command
- For each failed control surface, its time of failure and whether the control surface is currently operational or failed.
- The overall system state, such as operational, degraded, cooling, warm, off, etc.

This new understanding of the needs of the **AMS** stakeholders should result in new requirements. *Indeed, this is one of the primary objectives of doing the use case functional analysis – identify missing requirements*. Thus, we will add the following requirements to the requirements database and allocate them to the current use case (this occurs in the *Generate/Update System Requirements* task in the workflow shown in Figure 3). These new requirements are:

```
In response to a movement command from the AMS, the
system shall respond with a status message that
provides the operational status for each control
surface as well as the overall system operational
state.

The operational status reported to the AMS for each
control surface shall include its current commanded
position, its current measured position, the time
of measurement, and the time required to enact the
movement command for that control surface.
```

Add these as functional requirements in the **RequirementsAnalysisPkg > RequirementsPkg > FunctionalReqs** package. I named these requirements **FuncReq100** and **FuncReq101**. Be sure to add trace links from the Control Air Surface use case to these requirements.

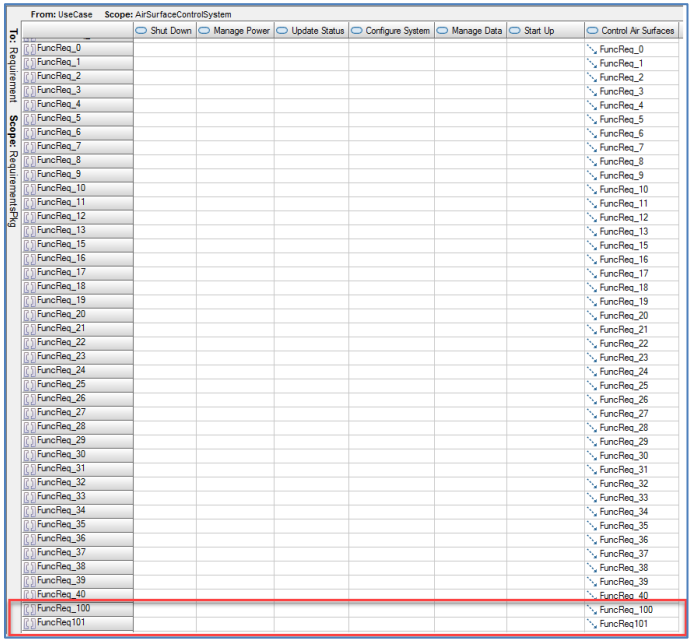


Figure 127: Adding trace links from new requirements to use case

Note that this status information is likely related to the **Update Status** use case, which periodically updates both the **AMS** and **Pilot Display** on a periodic basis. We expect that the **Update Status** use case will also have to send updates on the outcomes of periodic tests on hydraulics and power as well. These additional messages are not a part of this (the **Control Air Surfaces**) use case, so we can ignore them for now. When the **Update_Status** use case is analysed in a subsequent iteration, those additional needs will need to be merged together.

The data schema for the commanded positions and status placed are shown in Figure 128.

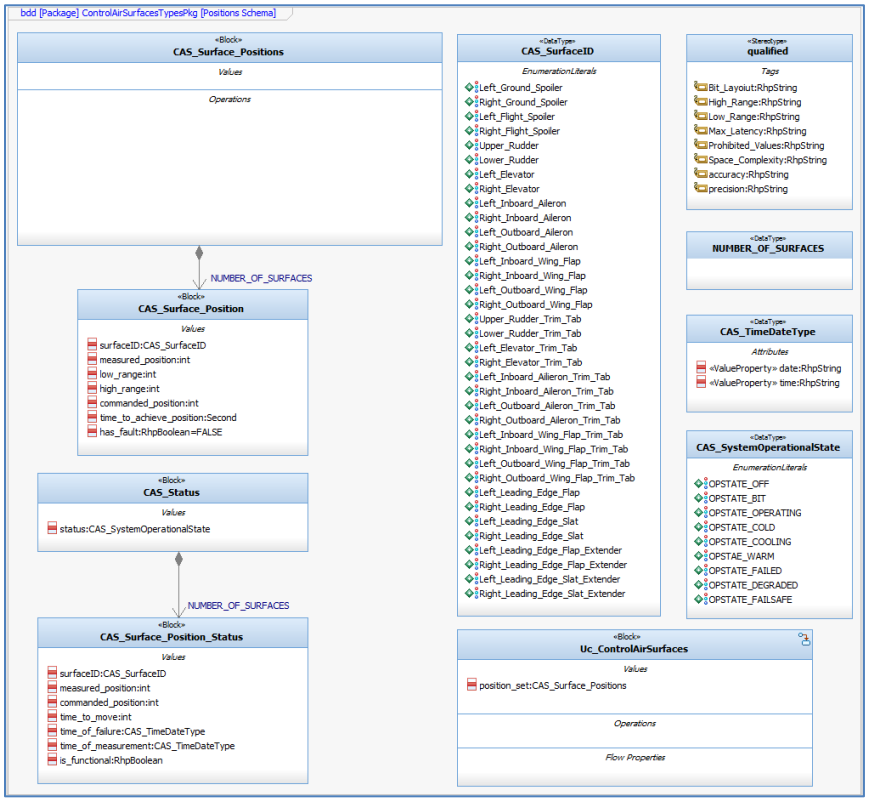
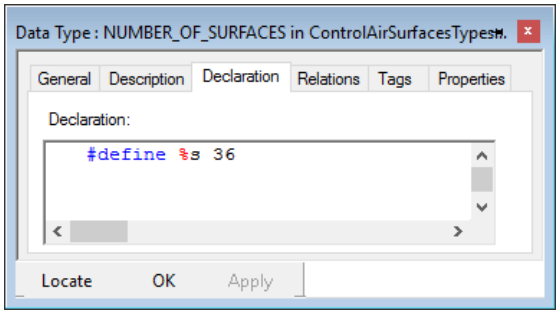


Figure 128: Control Air Surfaces Use Case Data Schema

CAS_Surface_Positions contains an array of the IDs and positions for all the surfaces. This data structure will be used later when we construct the executable state machine for the use case block as the means by which the **aCAS_AMS** actor sends commands to the use case block. The constant **NUMBER_OF_SURFACES** is defined as



and the **CAS_SurfaceID** type is an enumerated type listing all of the control surfaces.

What about surfaces that have trim tabs or extensible surfaces? These will be modeled as having unique **surfaceIDs** and so can be separately referenced.

The other interesting use of types is the use of **«qualified»** stereotype (from the HarmonySE profile) which adds the tags of **accuracy** (how close the measured value is to its true value) and **precision** (number of valid significant digits). These are important aspects of the specification and will drive downstream technology and design decisions. In this case, the requirements state:

```
The precision of the commanded values shall be +/-
0.1 degrees of angle or +/- .1 cm of distance. The
range of accuracy of commanded and measured
positions achieved shall be +/- 0.5 degrees or
angle of 0.5 cm of distance.
```

So the precision tags of **measured_position** and **commanded_position** attributes will be set to "+/- 0.1 degrees or cm". The accuracy tag for these attributes will be set to "+/- 0.05 degrees or cm."

We are not now interested in the bit mapping of the exact types that will be used in the developed system (the "physical data schema") but rather its logical properties. This is why this is called the "logical data schema". Physical data schema will be specified in the Handoff workflow.

7.4.4 Safety Analysis for Control Air Surfaces Use Case

As before, the approach we will take is to identify the hazards presented by the use case and create a fault tree analysis for each. In this case, there is only a single hazard we will consider: **Unable to Control Surfaces**. Right click on the **ControlAirSurfacesSafetyPkg** package in the browser and add a new FTA diagram. Name this diagram **FTA for Unable to Control Attitude**. As before, switch the project to the Dependability profile and back to SysML when this work is done.

Using the tools in the FTA diagram toolbar (**Hazard Condition**, **Required Condition**, **Transfer Operator**, **Logic Flow**, **AND** operator, and **OR** operator), draw the following FTA diagram.

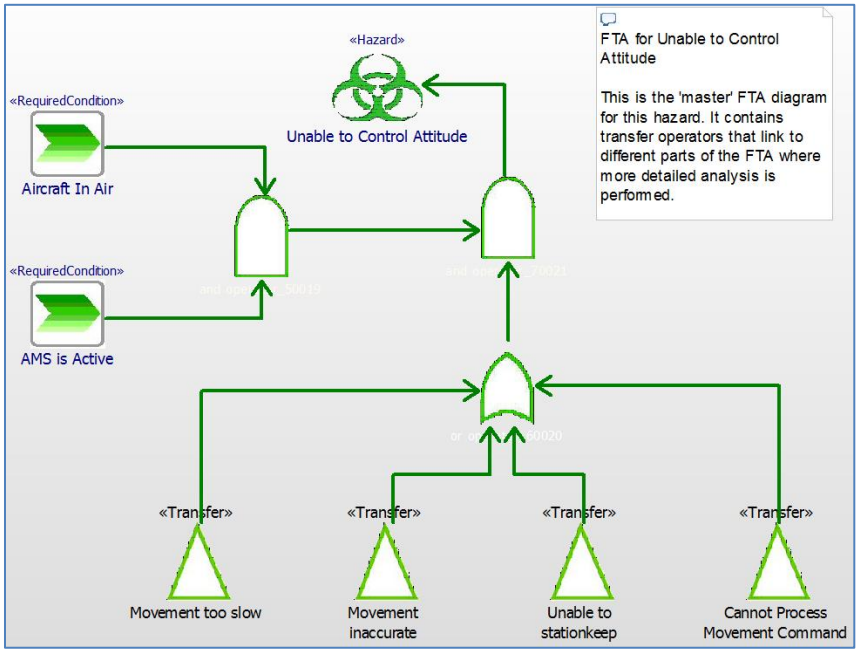


Figure 129: FTA Diagram for Unable to Control Attitude

The transfer operators refer to analyses on other diagrams that logically feeds into this diagram. Let's create those other diagrams now.

In the browser, right-click *FunctionalAnalysisPkg > ControlAirSurfacesPkg > ControlAirSurfacesSafetyPkg > FTADiagrams* and select *Add New FTA Diagram*. Name this diagram **Movement Too Slow FTA**. Before we elaborate the diagram, let's link it to the proper transfer operator. In the first FTA diagram, right-click the transfer operator **Movement too slow** and select Add *New -> Hyperlink*. Click on the *Target Name* radio button to show the name of the master FTA diagram and use the drop down list to select that original FTA diagram from the list. Click *OK* and then Click *OK* again to forge the link.

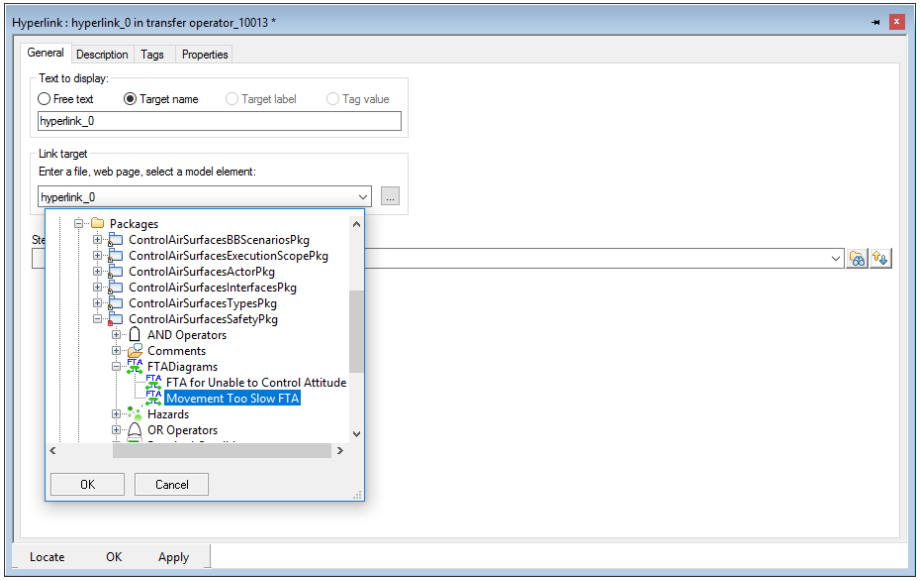


Figure 130: Linking FTA diagrams with transfer operators

Now, when you right-click on the transfer operator, there will be an option to select *Hyperlink > Movement Too Slow FTA*. Use that hyperlink now to open and navigate to the empty FTA diagram.

At the top of this diagram add a new transfer operator and name it **Movement Too Slow Outcome**. Repeat the hyperlink steps above to link this operator with the master diagram **FTA for Unable To Control Attitude**. We now have bi-directional hyperlink navigation between the two diagrams. We will do the same for the other three FTA to come to a have a linked set[14].

Now fill out the rest of the diagram as shown in Figure 131.

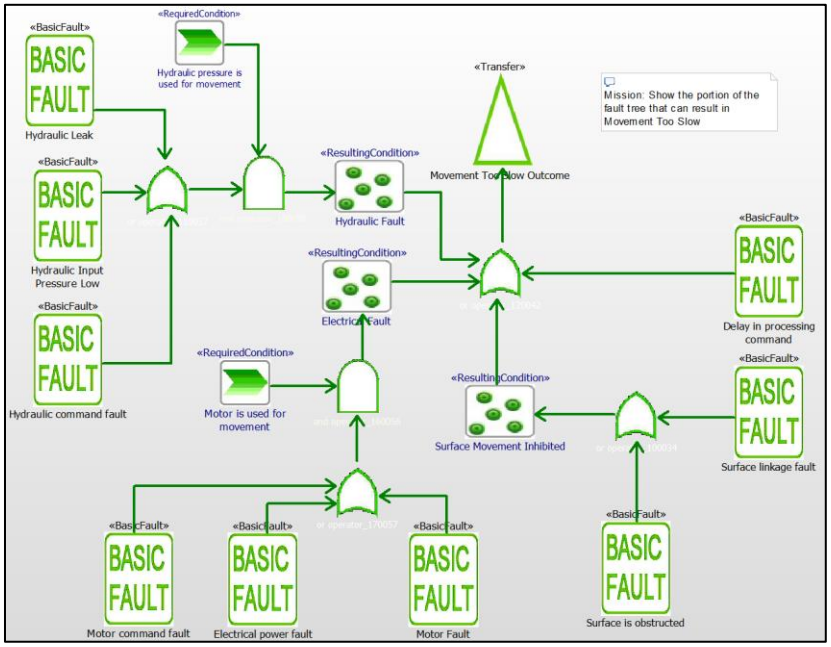


Figure 131: Movement Too Slow FTA

The way to interpret Figure 131 is that an outcome of **Movement Too Slow** can result any of from independent conditions:

1. The system is using hydraulics for movement AND (either the input hydraulic pressure is too low OR the system has a hydraulic leak OR there is a fault in determining the actual movement command), OR
2. The system is using an electric motor for movement AND (there is a fault in either the electrical power system OR in the motor itself OR there is a fault in determining the actual movement command), OR
3. Something is obstructing the free movement of the control surface OR there is a fault in the mechanical linkage of the surface to the moving force, OR
4. The system was delayed in processing the command

These ORed conditions that can manifest a hazard are commonly known as *cut-sets*. The purpose in doing this analysis is to identify where safety control measure should be added to improve the safety of the system.

[14] You can accomplish the same thing using *Resulting Condition* operators as well. This is normally used for reusable causality "subroutines" of small interactions resulting in a condition that will be reused in many FTAs while *Transfer* operators are normally used in the decomposition of a single FTA.

To create the next few diagrams, you *can* reuse fault elements, such as **Hydraulic Input Pressure Low** by dragging them from the browser onto a new diagram. Remember, however, you *must not* reuse the logical operators (*AND, OR, NOT, NAND, NOR, XOR,* or *Transfer*); these operators have identity and if you attempt to reuse them, while the diagrams with look ok, you will mess up the causality relations.

Using a similar approach, create the next three diagrams and link them bi-directionally with hyperlinks to the original **FTA For Unable to Control Attitude** diagram.

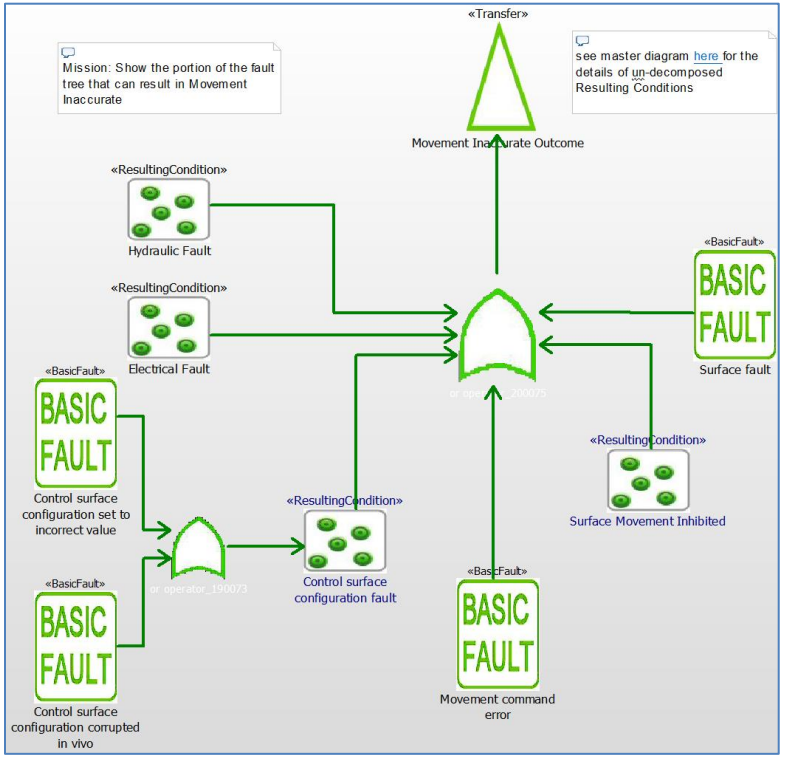


Figure 132:Movement Inaccurate FTA

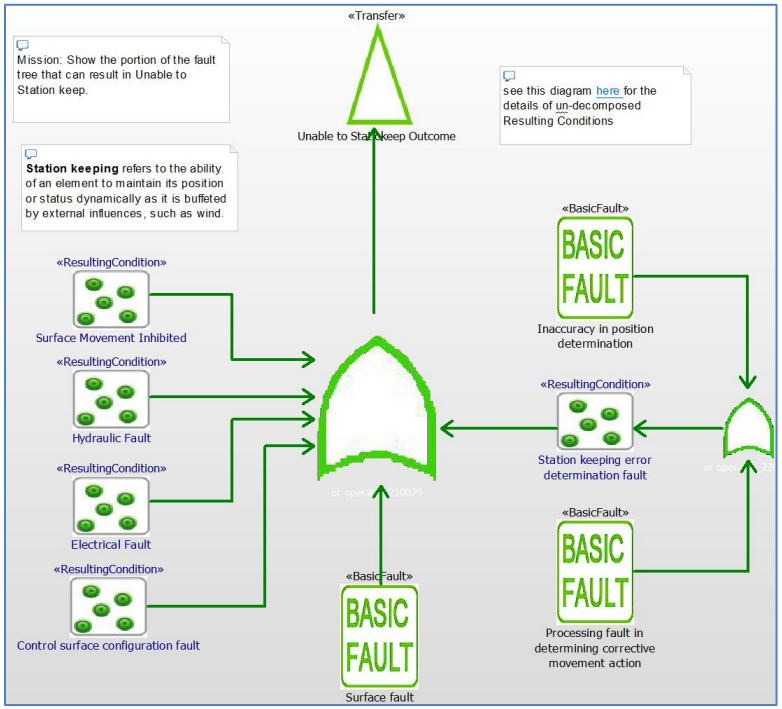


Figure 133: Unable to Station keep FTA

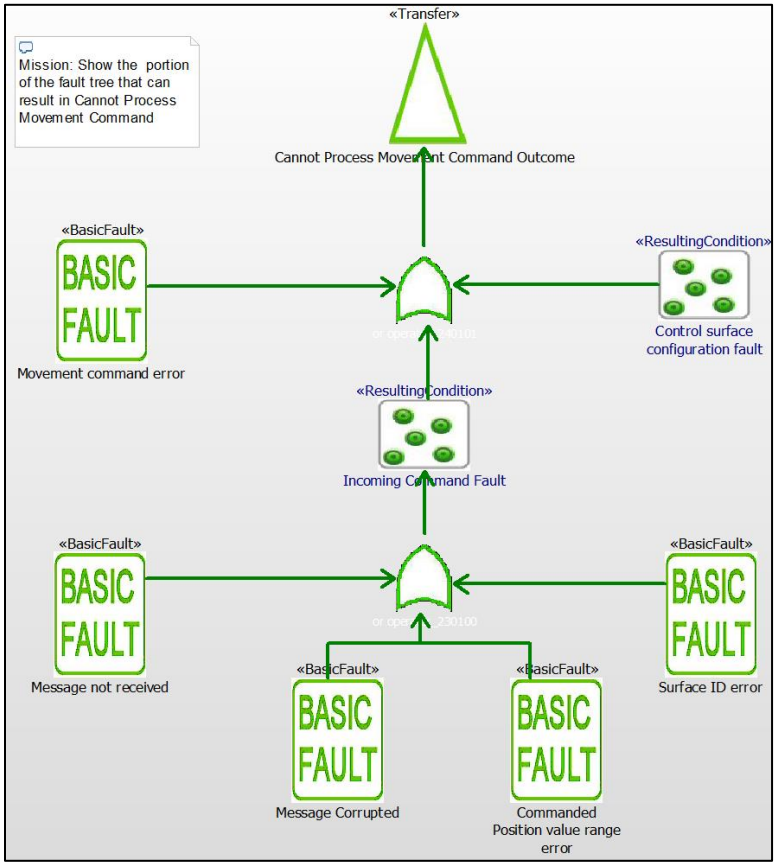


Figure 134: Cannot Process Movement Command FTA

> A hint for resizing elements (especially the logic operators) on the FTA diagrams – hold the SHIFT key down while resizing to maintain the same aspect ratio.

The next step is to use the FTA diagram to reason about the safety and reliability of the system; especially the what, when and where of the use of safety control measures. For the sake of brevity, we will only update one of the five FTA diagrams we just created. In a real system design, you would perform this work for all FTAs.

All safety control measures either make the hazard condition less likely or less severe. The easiest to model is the former; this approach leads to the identification of *ANDing conditions* for the fault logic flows. They are named such because both, the original fault AND the fail of the safety control measure must occur, in order to manifest a fault. The likelihood of two independent faults is the product of the likelihood of each separate fault. So if the likelihood of fault A occurring is 10% (0.10) and the likelihood of the safety measure failing is 5% (0.05), then the likelihood of both occurring is 0.5% (0.005). The identification of the need for safety measures then results in *safety requirements* which are added to the requirements specification and allocated to the use case under analysis.

In the case of this system, there are certainly opportunities to add safety measures. Let's consider each of the *OR*ed conditions in Figure 132 (**Movement Inaccurate**) separately. If we make each of the *OR*ed conditions less likely, we improve the system safety with respect to their underlying fault conditions. To improve readability, we'll create a separate diagram to analyze each *OR*ed condition rather than create a (much) larger single diagram.

First, let's consider the **Hydraulic Fault** Resulting Condition. In this case, the system is relying on the aircraft hydraulics for pressure and then distributing that pressure internally to move some of the control surfaces. Due to weight limitations (separate analysis, not shown here), we cannot create a fully redundant hydraulic system, so we decide that it is enough to detect

that low input pressure or internal leaks and report them to the AMS. The other **Basic Fault** is that the hydraulic command for movement is incorrect to achieve the desired position. Add the new Basic **Faults**, **logic operators**, **Safety Requirements**, and **traceToReqs** relations to the model. Figure 135 shows the resulting FTA.

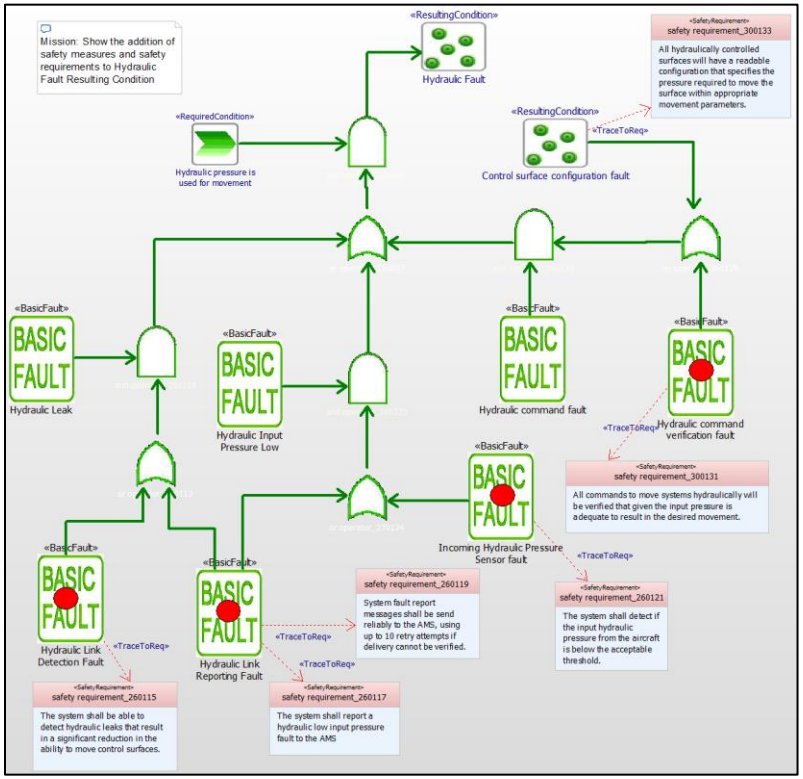


Figure 135: Hydraulic Fault Safety Measures FTA

We can see that we've (indirectly) added safety measures because the new **Basic Faults** (indicated with the red dots overlaid onto the **Basic Fault**[15]) refer to what happens when those safety measures fail. You can also see that we've added new safety requirements with appropriate trace links:

- The system shall be able to detect hydraulic leaks that result in a significant reduction in the ability to move control surfaces.
- The system shall report a hydraulic low input pressure fault to the AMS
- System fault report messages shall be send reliably to the AMS, using up to 10 retry attempts if delivery cannot be verified.
- The system shall detect if the input hydraulic pressure from the aircraft is below the acceptable threshold.
- All commands to move systems hydraulically shall be verified that given the input pressure is adequate to result in the desired movement.
- All hydraulically controlled surfaces shall have a readable configuration that specifies the pressure required to move the surface within appropriate movement parameters.

These requirements must be added to the requirements specification. In the **RequirementsAnalysisPkg > RequirementsPkg** add a new package named **SafetyReqs** and add them there. Add trace links to the **Control Air Surfaces** use case. This is most easily done in the **AirSurfaceControlSystemUseCaseRequirementsMatrix** in the **RequirementsAnalysisPkg** package. The identification of the safety measures and corresponding requirements is the point of doing this safety analysis within the use case functional analysis.

Next, let's apply the same reasoning to the Resulting Condition of **Electrical Fault** (Figure 136).

[15] I'm not recommending putting red dots on the icons – this is just to show them in this Deskbook.

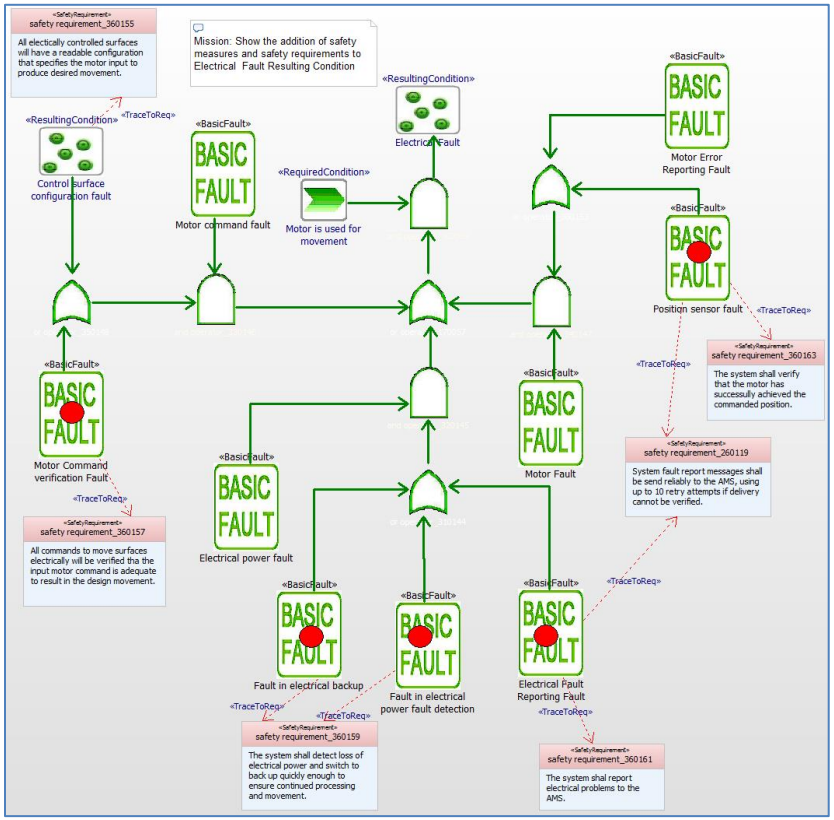


Figure 136: Electrical Fault Safety Measures

This work has resulted in several more requirements that will be added to the specification and linked to the use case.

Now let's look at the condition of **Control Surface Configuration Fault**. In this case, we'll add safety measures to verify the command by returning the command value, once set, to the **Maintainer** actor for verification. We'll also protect the configuration data by storing it redundantly, and check that the configuration data is not corrupted prior to its use. If found to be corrupted, the corresponding control surface is marked as disabled, and the **AMS** is notified. That results in the FTA shown in Figure 137:

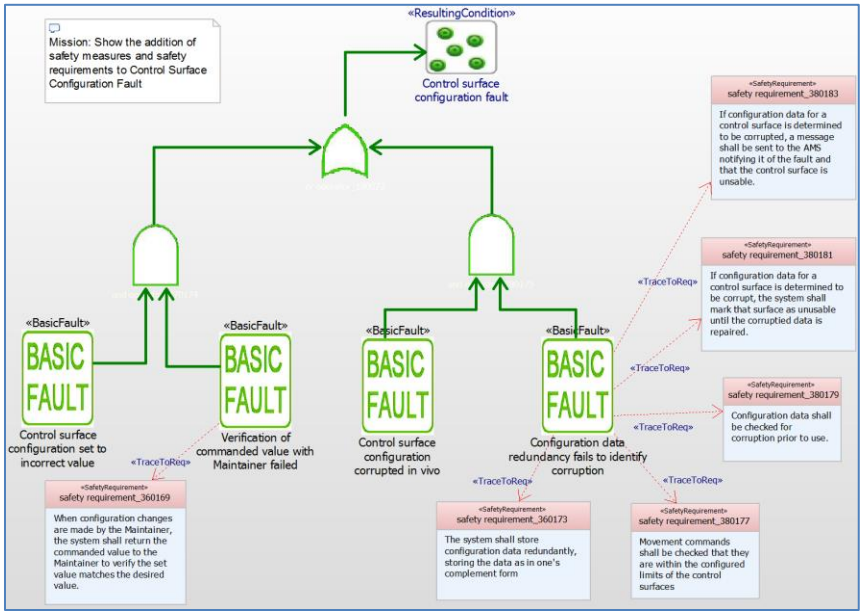


Figure 137: Control Surface Configuration Fault Safety measures

Movement Command Error is considered a basic fault but really is that the command from the **AMS** was corrupted, referenced an invalid *Control Surface ID*, or commanded a position that was out of its range (as determined by the configuration for that control surface). As for **Surface fault**, that refers to mechanical damage to the control surface itself. In this case, we increase safety by improving the reliability of the control surface. We'll handle that by specifying the MTBF of the control surface materials and provide a specification of resistance to impact force. This results in an update to Figure 132. The result is shown in Figure 138.

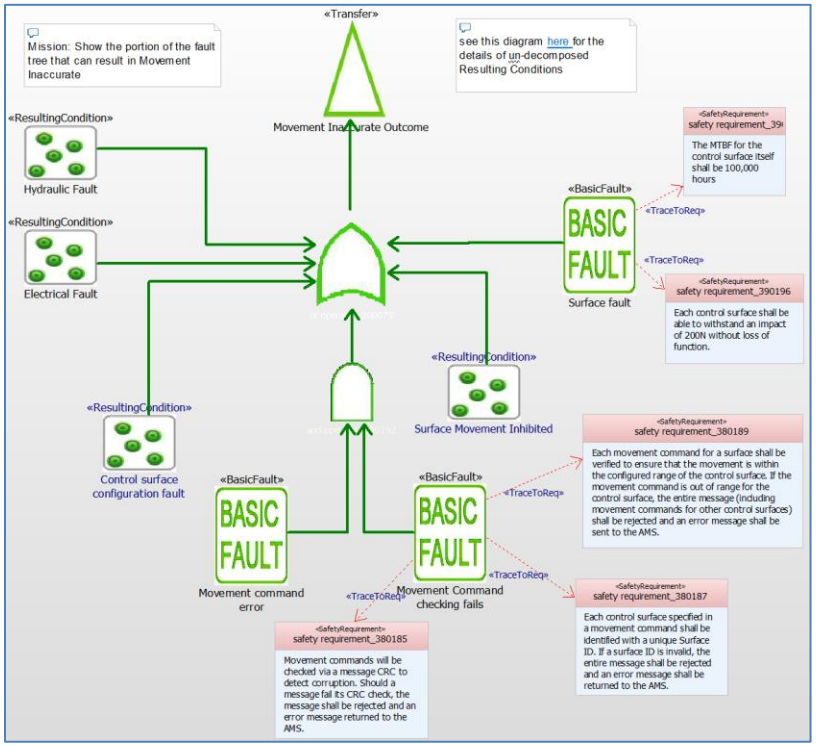


Figure 138: Movement Command Error Safety Measures FTA

Lastly, for the **Surface Movement Inhibited** Resulting Condition we have decided that proper maintenance should identify and repair these concerns adequately, so no new requirements are necessary.

As a result of the fault tree analysis, we have identified a number of safety concerns and identified a total of 22 new safety requirements which are added to the requirements specification and allocated to the **Control Air Surfaces** use case. These should be placed in the **RequirementsPkg**, traced to the **Control Air Surfaces** use case, modeled in (new) scenarios and will be represented in the state machine (coming up next).

7.4.5 Create the Control Air Surfaces Use Case State Machine (and execute it too!)

The next step in the process workflow (Figure 115) is to construct the state machine. The best way to construct such a state machine is incrementally. Although the process flow in Figure 115 shows the state machine being created and then in the next step being executed, actual practice has shown that it is best to construct the state machine in a series of small steps and use execution at each step to ensure that it is right *so far* before adding more state machine elements. This is such an important idea, let's call it out in a side note:

> The best way to construct a possibly complex state machine is in a series of small steps – called *nanocycles* – wherein the state machine correctness is verified at the end of each step. These steps typically take between 10 and 60 minutes to complete.

In this example, we will just show the state machine in multiple stages:

1. Command received results in movement of a set of control surfaces
2. Commanded movement is out of range
3. Movement is inaccurate or too slow
4. Faults are detected at run-time in specific surfaces

We'll start with a simple *sunny day* case and get that state machine running (simulated, of course). Then, we'll progressively add more error states, conditions, and events. At each stage, we'll add state behavior to model some more requirements.

7.4.5.1 Stage 1: Sunny day control surface movement

Let's add a state machine to the use case block **Uc_ControlAirSurfaces**. Figure 139 shows the state machine. It looks trivially easy – so, it should be a snap to get it to run.

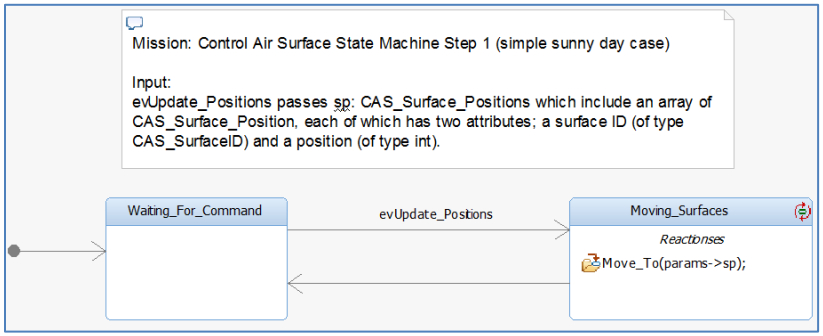
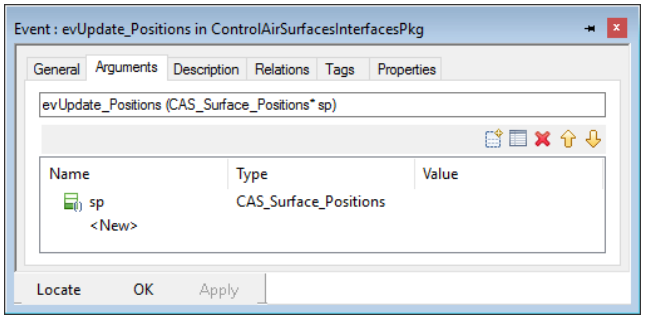


Figure 139: Control Air Surfaces use case state machine step 1

It turns out that it's not quite so trivially easy. The event **evUpdate_Positions** passes the set of surface command positions using the **CAS_Surface_Positions** data structure shown in Figure 128. So we'll have to add the parameter to the event and write a small number of functions to manipulate that data in order to get the execution working.

The act of drawing the event on the state machine creates the event. To add the parameter to the event, locate the event in the browser at **FunctionalAnalysisPkg > ControlAirSurfacePkg > ControlAirSurfacesInterfacePkg**. Double click on the **evUpdate_Pos** event to open its *Features* dialog. On the *Arguments* tab, add an argument **sp**. Use the type pull down list followed by the *Select* option to select its type **CAS_Surface_Positions**.

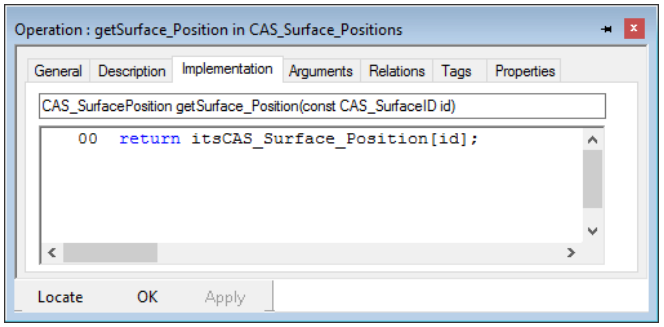


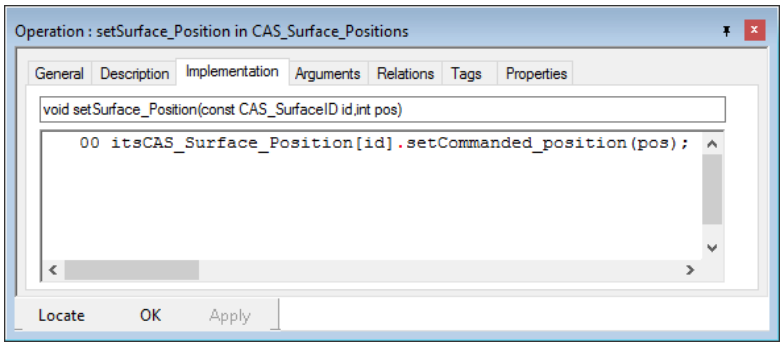
We need to add some actions to support getting and setting these attributes. Because the multiplicity of the composition relation between the **CAS_Surface_Positions** block and the **CAS_Surface_Position** block (shown in Figure 128) is more than one and fixed, Rhapsody generates an array to hold the values. This is suitable for our purposes (simulation).

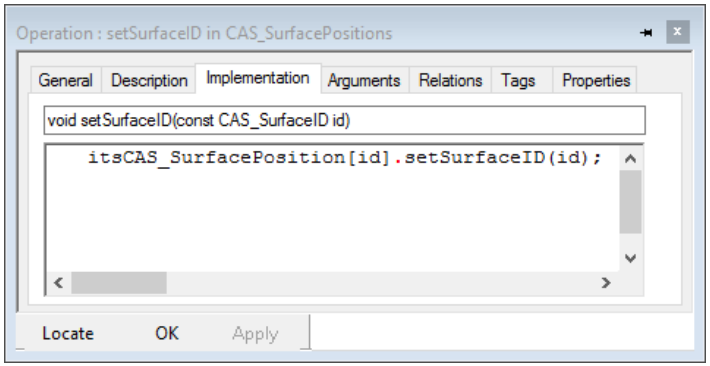
We will define three operations to access the individual elements of the array:

- getSurface_Position(id: CAS_SurfaceID): CAS_Surface_Postion
- setSurface_Position(id: CAS_SurfaceID, pos:int)
- setSurfaceID(id:CAS_SurfaceID): void

The next three figures provide the implementation of those operations:







These operations allow an element with a pointer to a **CAS_Surface_Positions** instance to access individual surface position values.

Now, we will "instrument the actor" by adding state machine behavior to the **aCAS_AMS** actor block, so that it sends the event (along with values for the **sp** argument) to the use case block.

First, let's add four value properties to the actor block, named **zero**, **positionSet1**, **positionSet2**, **positionSet3**. Each of these should be of type **CAS_SurfacePositions**. Then create the state machine for the **aCAS_AMS** actor block. We will add new events that we will use to drive the simulation, **evZero**, **evPos1**, **evPos2**, and **evPos3**. Each will cause a transition to a *Send Event* that sends the **evUpdate_Positions** with the appropriate argument, to the port **pUc_ControlAirSurfaces**. At run time, this port will be connected to a corresponding port on the use case block, so it will receive these events and arguments to act on.

For example, the **evZero** event will active the *Send Action* with the event **evMovement** carrying the data **&zero** to the port **pUc_ControlAirSurfaces**:

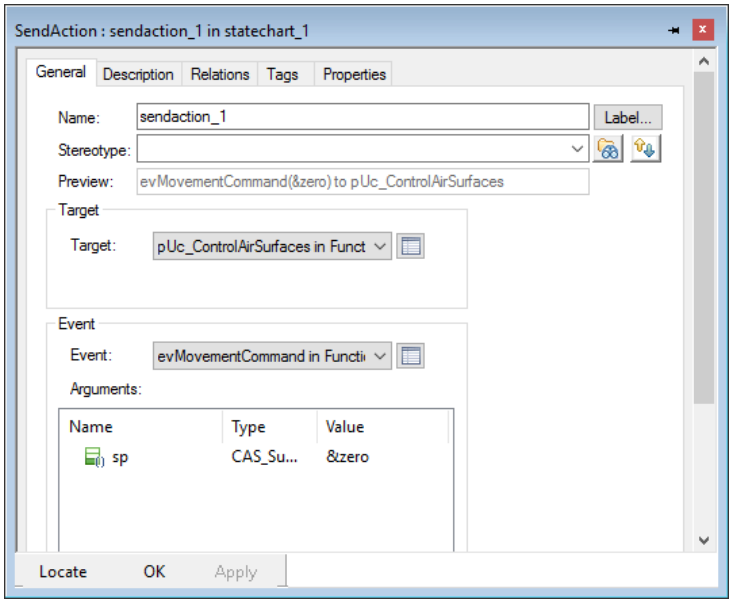


Figure 140: Adding parameters to the Send Action invoked by the evZero event

We'll repeat this for the other three events, each sending a different attribute. When complete, the state machine for the actor block should look like Figure 141.

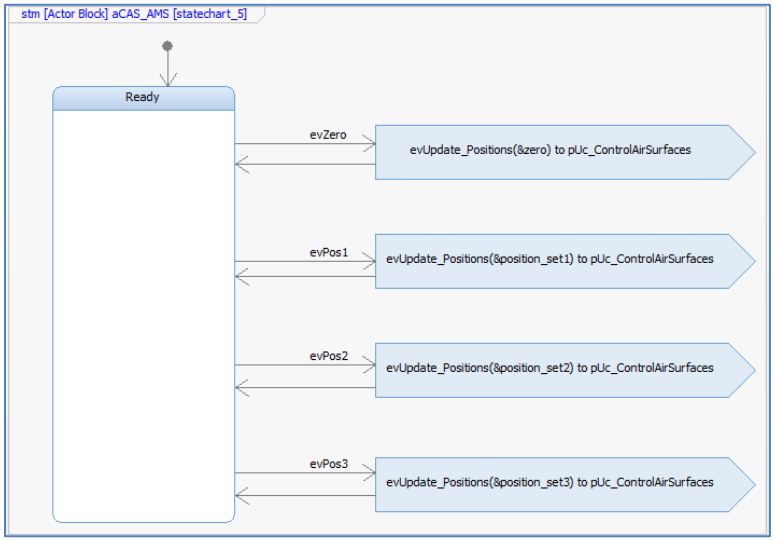


Figure 141: aCMS_AMS State Machine step 1

Note that we use "&" before the parameter name in Figure 140 because by default Rhapsody sends complex structures as a pointer, so we must pass the address of the attribute to get to its values.

To complete the work on the actor block, we should assign values to the control surface positions that we're going to send to the use case block. We'll do that in the **setUpPositions()** operation shown in Figure 141.

In the browser, right click on the **aCAS_AMS** actor block and select *Add New -> Operation*. Name this operation **Setup_Positions**. On the implementation tab of the features dialog, add the following implementation:

```
// set up the positions sets up to the Right_Inboard_Aileron (first 10
surfaces)
zero.setSurface_Position(Left_Ground_Spoiler, 0);
zero.setSurfaceID(Left_Ground_Spoiler);
zero.setSurface_Position(Right_Ground_Spoiler, 0);
zero.setSurfaceID(Right_Ground_Spoiler);
zero.setSurface_Position(Left_Flight_Spoiler, 0);
zero.setSurfaceID(Left_Flight_Spoiler);
zero.setSurface_Position(Right_Flight_Spoiler, 0);
zero.setSurfaceID(Right_Flight_Spoiler);
zero.setSurface_Position(Upper_Rudder, 0);
zero.setSurfaceID(Upper_Rudder);
zero.setSurface_Position(Lower_Rudder, 0);
zero.setSurfaceID(Lower_Rudder);
zero.setSurface_Position(Lower_Rudder, 0);
zero.setSurfaceID(Left_Elevator);
zero.setSurface_Position(Left_Elevator, 0);
zero.setSurfaceID(Right_Elevator);
zero.setSurface_Position(Right_Elevator, 0);
zero.setSurfaceID(Left_Inboard_Aileron);
zero.setSurface_Position(Left_Inboard_Aileron, 0);
zero.setSurfaceID(Right_Inboard_Aileron);
zero.setSurface_Position(Right_Inboard_Aileron, 0);

// set up postionsSet1
position_set1.setSurface_Position(Left_Ground_Spoiler, 1);
position_set1.setSurfaceID(Left_Ground_Spoiler);
position_set1.setSurface_Position(Right_Ground_Spoiler, 2);
position_set1.setSurfaceID(Right_Ground_Spoiler);
position_set1.setSurface_Position(Left_Flight_Spoiler, 3);
position_set1.setSurfaceID(Left_Flight_Spoiler);
position_set1.setSurface_Position(Right_Flight_Spoiler, 4);
position_set1.setSurfaceID(Right_Flight_Spoiler);
position_set1.setSurface_Position(Upper_Rudder, 5);
position_set1.setSurfaceID(Upper_Rudder);
position_set1.setSurface_Position(Lower_Rudder, 6);
position_set1.setSurfaceID(Lower_Rudder);
position_set1.setSurfaceID(Left_Elevator);
position_set1.setSurface_Position(Left_Elevator, 7);
position_set1.setSurfaceID(Right_Elevator);
position_set1.setSurface_Position(Right_Elevator, 8);
position_set1.setSurfaceID(Left_Inboard_Aileron);
position_set1.setSurface_Position(Left_Inboard_Aileron, 9);
position_set1.setSurfaceID(Right_Inboard_Aileron);
position_set1.setSurface_Position(Right_Inboard_Aileron, 10);

// now for postionSet2
position_set2.setSurface_Position(Left_Ground_Spoiler, -1);
position_set2.setSurfaceID(Left_Ground_Spoiler);
position_set2.setSurface_Position(Right_Ground_Spoiler, -2);
position_set2.setSurfaceID(Right_Ground_Spoiler);
position_set2.setSurface_Position(Left_Flight_Spoiler, -3);
position_set2.setSurfaceID(Left_Flight_Spoiler);
position_set2.setSurface_Position(Right_Flight_Spoiler, -4);
position_set2.setSurfaceID(Right_Flight_Spoiler);
position_set2.setSurface_Position(Upper_Rudder, -5);
position_set2.setSurfaceID(Upper_Rudder);
position_set2.setSurface_Position(Lower_Rudder, -6);
position_set2.setSurfaceID(Lower_Rudder);
position_set2.setSurfaceID(Left_Elevator);
position_set2.setSurface_Position(Left_Elevator, -7);
position_set2.setSurfaceID(Right_Elevator);
position_set2.setSurface_Position(Right_Elevator, -8);
position_set2.setSurfaceID(Left_Inboard_Aileron);
position_set2.setSurface_Position(Left_Inboard_Aileron, -9);
position_set2.setSurfaceID(Right_Inboard_Aileron);
```

```
position_set2.setSurface_Position(Right_Inboard_Aileron, -10);


// and some out of range values for position_set3
position_set3.setSurface_Position(Left_Ground_Spoiler, 100);
position_set3.setSurfaceID(Left_Ground_Spoiler);
position_set3.setSurface_Position(Right_Ground_Spoiler, 45);
position_set3.setSurfaceID(Right_Ground_Spoiler);
position_set3.setSurface_Position(Left_Flight_Spoiler, -100);
position_set3.setSurfaceID(Left_Flight_Spoiler);
position_set3.setSurface_Position(Right_Flight_Spoiler, -50);
position_set3.setSurfaceID(Right_Flight_Spoiler);
position_set3.setSurface_Position(Upper_Rudder, 47);
position_set3.setSurfaceID(Upper_Rudder);
position_set3.setSurface_Position(Lower_Rudder, -60);
position_set3.setSurfaceID(Lower_Rudder);
position_set3.setSurfaceID(Left_Elevator);
position_set3.setSurface_Position(Left_Elevator, -33);
position_set3.setSurfaceID(Right_Elevator);
position_set3.setSurface_Position(Right_Elevator, -92);
position_set3.setSurfaceID(Left_Inboard_Aileron);
position_set3.setSurface_Position(Left_Inboard_Aileron, 150);
position_set3.setSurfaceID(Right_Inboard_Aileron);
position_set3.setSurface_Position(Right_Inboard_Aileron, -1500);
```

This implementation only sets the ids and position values for the first 10 surfaces. If you want to be more complete and set all 36, feel free to do so.

The last thing we need to create before we can run the model is to implement the **Move_To(params->sp)** operation used in Figure 139.

> Rhapsody uses the slightly odd **params** syntax to pass event arguments. To reference a value passed as an argument in an event, Rhapsody creates a *struct* called **params** and makes all the pass arguments fields of that *struct*. See Section Appendix: Passing Data Around in Rhapsody for C++12 for more details on this.

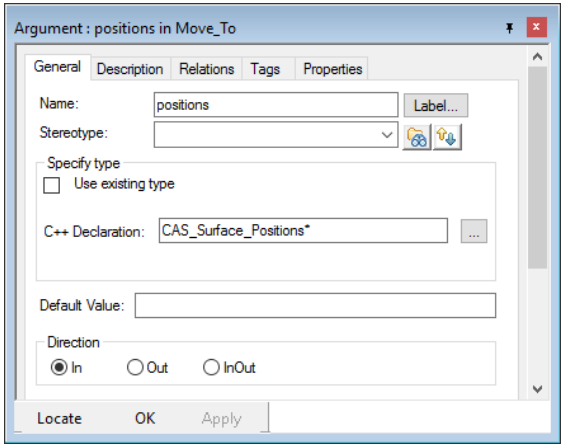
To do this implementation, we will create smaller functions to assist. **Move_To ()** will call **Set_Position()** for each surface. **Set_Position()**, in turn, sets a local attribute **position_set** (of type **CAS_Surface_Positions**) with the passed values. For debugging, we'll also print the values out to standard output so that we can visually see what's going on.

1. Create the **position_set** attribute

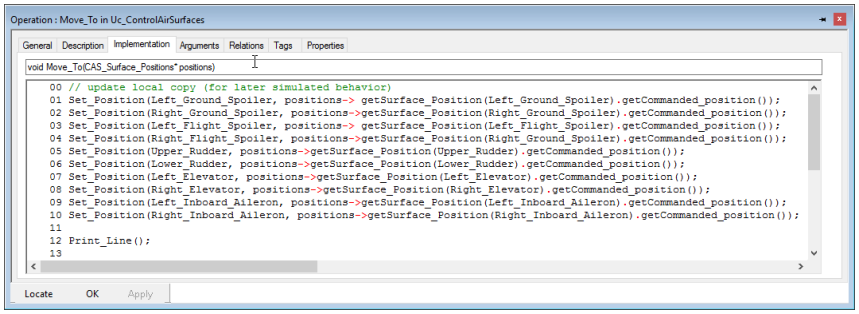
In the browser, right click on the use case block **Uc_ControlAirSurfaces** and select *Add New -> Value Property*. Name the attribute **position_set** and specify its type as **CAS_Surface_Positions**. This attribute will hold the positions of the surfaces.

2. Create the M**ove_To** operation

In the browser, right click on the use case block **Uc_ControlAirSurfaces** and select *Add New -> Operation.* Name the operation **Move_To**. In the aguments tab of the operation features dialog, add an argument **positions**. Double click on the argument name to open its *Features* dialog. In this *Features* dialog, deselect the *Use Existing Type* checkbox and type in: **CAS_SurfacePositions*** as the declaration (don't omit the trailing '*' which identifies the element as a pointer to a type):



Click *OK* to return to the operation features dialog. In the implementation tab of the **Move_To** features dialog, enter the following implementation:



Since the image is a bit small, here is the implementation a bit larger:

```
// update local copy (for later simulated behavior)
Set_Position(Left_Ground_Spoiler, positions->
 getSurface_Position(Left_Ground_Spoiler).getCommanded_position());
Set_Position(Right_Ground_Spoiler, positions->
 getSurface_Position(Right_Ground_Spoiler).getCommanded_position());
Set_Position(Left_Flight_Spoiler, positions->
 getSurface_Position(Left_Flight_Spoiler).getCommanded_position());
Set_Position(Right_Flight_Spoiler, positions->
 getSurface_Position(Right_Ground_Spoiler).getCommanded_position());
Set_Position(Upper_Rudder, positions->
 getSurface_Position(Upper_Rudder).getCommanded_position());
Set_Position(Lower_Rudder, positions->
 getSurface_Position(Lower_Rudder).getCommanded_position());
Set_Position(Left_Elevator, positions->
 getSurface_Position(Left_Elevator).getCommanded_position());
Set_Position(Right_Elevator, positions->
 getSurface_Position(Right_Elevator).getCommanded_position());
Set_Position(Left_Inboard_Aileron, positions->
 getSurface_Position(Left_Inboard_Aileron).getCommanded_position());
Set_Position(Right_Inboard_Aileron, positions->
 getSurface_Position(Right_Inboard_Aileron).getCommanded_position());

Print_Line();
```

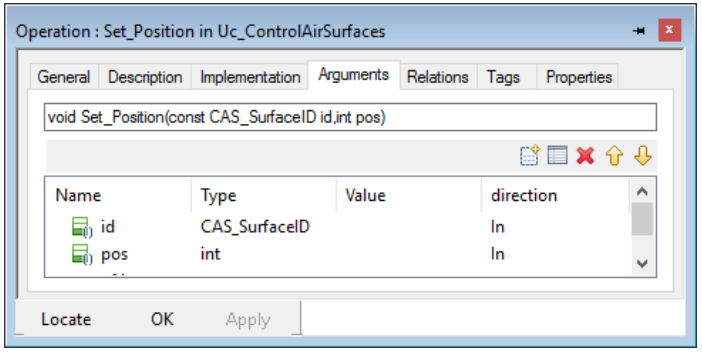
This operation calls **Set_Position** for each (of the first 10) positions and then finishes with a call to **Print_Line()** to send an extra line feed to standard output.

3. Create **Set_Position** operation

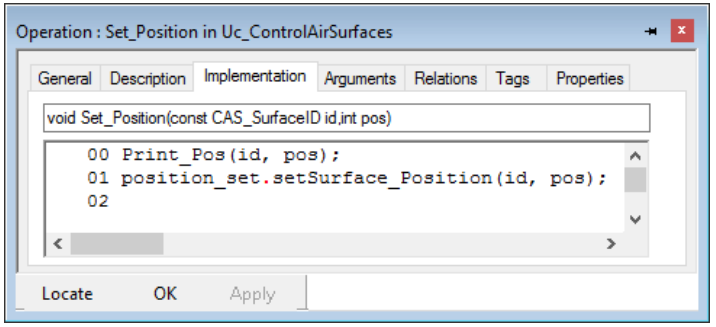
This and the **Print_Pos** operation are here to assist in the control and visualization of the simulation. As such, they do not represent requirements. The HarmonySE profile contains a stereotype to mark such elements:

«nonNormative». It indicates elements that do not represent a part of a specification per se and so do not represent requirements or design. All such elements should be so marked.

Repeat the previous procedure for adding a new operation to the **Uc_ControlAirSurfaces** block. This type, name the operation **Set_Position** and give it two parameters. The first, **id**, is of type **CAS_SurfaceID**. The second, **pos** (of type **int**) is the position value to set.



In the implementation tab, add the implementation:



4. Add the **Print_Pos** operation

This operation is meant to print the values for debugging. As before, add the new operation to the use case block and give it the same parameter list as the **Set_Position()** function. For implementation, just add the following:

```
std::cout << "Surface " << id << " at position " << pos << std::endl;
```

I'm implementing this model with the Cygwin compiler. It requires the std:: prefix on cout and endl applicators. If you're using another compiler, such as older versions of Microsoft Visual C++, you might need to use the line without the prefix:

```
cout << "Surface " << id << " at position " << pos << endl;
```

5. Add the **Print_Line** operation

This is a very simple function that just adds a blank line between sets of outputs. Add the new operation as before but don't give it any arguments. Specify the implementation as

```
std::cout << std::endl;
```

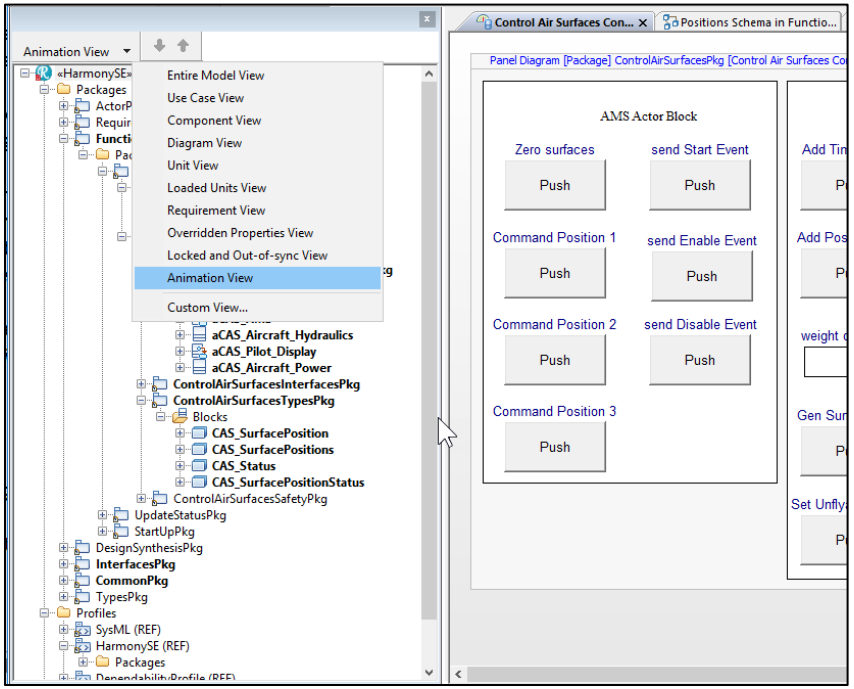
We are now ready to run!

Running the model

If you've entered all the model correctly so far, and Rhapsody is correctly configured to operate with your compiler, clicking on the *GMR* (*Generate/Make/Run*) button (or *Simulate*) button with generate code the model code, run the compiler and linker to generate an executable, and then run that executable.

Run the model and click on the *Go Idle* on the *Execution* control toolbar of Rhapsody. We'll now open three debugging windows in Rhapsody so we can view the execution. The instance statecharts for the running instances of the **Uc_ControlAirSurfaces** and **aCAS_AMS** blocks, and an animated sequence diagram.

To open the instance statecharts, in the browser, navigate to those blocks. In each you should see an Instances group under the block. Click on the '+' to see the instances and then right click on the instance and select *Open Instance Statechart* for each. Rhapsody has a useful filter for the browser under such circumstances. When a model is simulating, Rhapsody provides an *Animation Browser Filter* to show only elements related to the simulation.



To create an animated sequence diagram, click on the Rhapsody *Tools* menu and select *Animated Sequence Diagram*. Rhapsody will present you with a mini-browser to select the sequence diagram to use as a basis. Select one of the sequence diagrams we've created earlier.

I recommend you open an event insertion window. To do this, you can click on the *event generator* button on the execution toolbar.

Now arrange the windows how you like. I prefer an arrangements such as Figure 142.

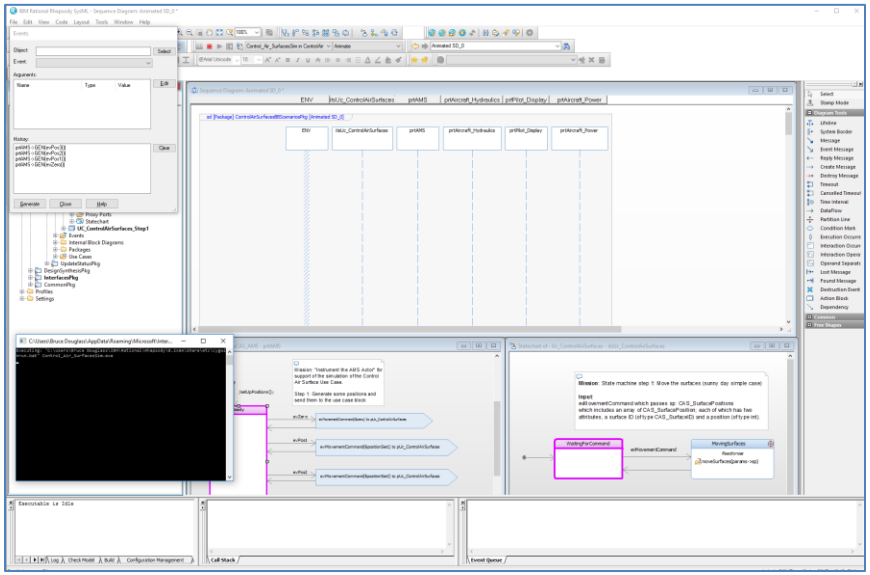


Figure 142: Running Step 1

In the event generator dialog, select the **prtAMS** instance (it may be at the bottom of the list of instances created). Select an event to run, say **evPos3** and click on *Generate*. If the model is not now running, you can click on either the *Go* or *Go Idle* buttons to step the model through the processing of the event. Rhapsody will run the model and show you the current and last states of the state machines, the messages on the sequence diagram, and the output sent to your computer's standard output.

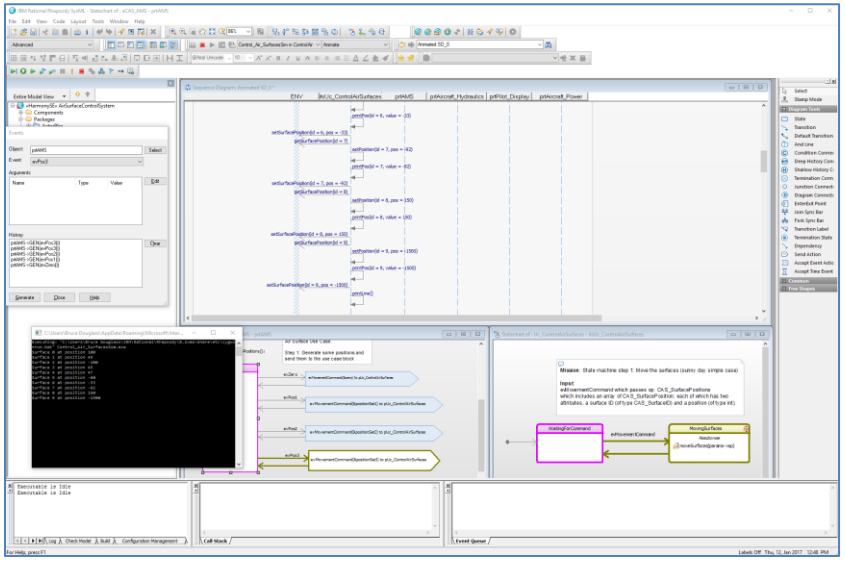


Figure 143: Event processing

You can explore the model execution by sending the events in different orders to satisfy yourself that it is properly representing the requirements you've modeled.

This may seem like a lot of work but most of the simulation support work is done and we can spend more mental focus on adding the remaining requirements.

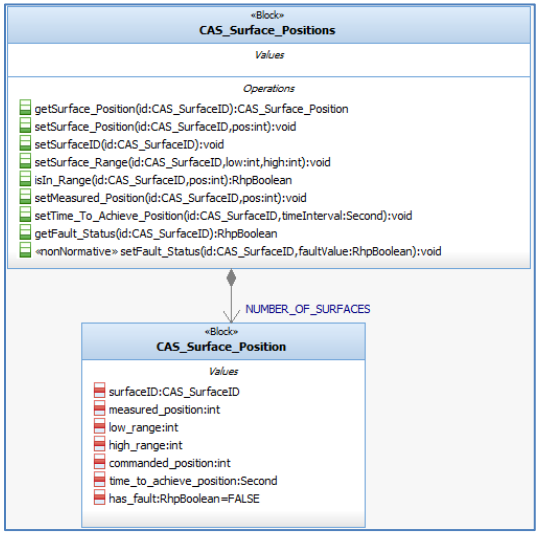
There are more requirements to add, so we'll do some more nanocycle iterations:

Step 1: Receive movement command and enact it in simplest case (complete)
Step 2: Validate command ranges
Step 3: Validate resulting movement and timing
Step 4: Handling requirements about warm and cold restarts
Step 5: Manage "flyable" operational state with surface faults

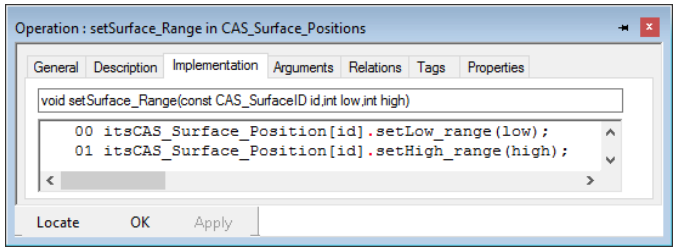
Step 2: Validate command and validate resulting movement and timing

Adding the requirements around validating the commands is straightforward but requires a number of small additions to the model.
=

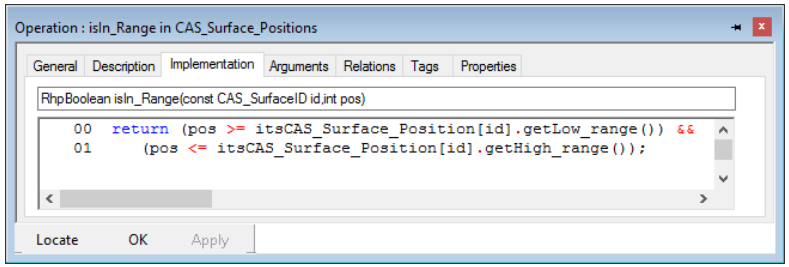
- We'll use the **CAS_Surface_Position** value properties **low_range**, **high_range**, and **measured_position.**



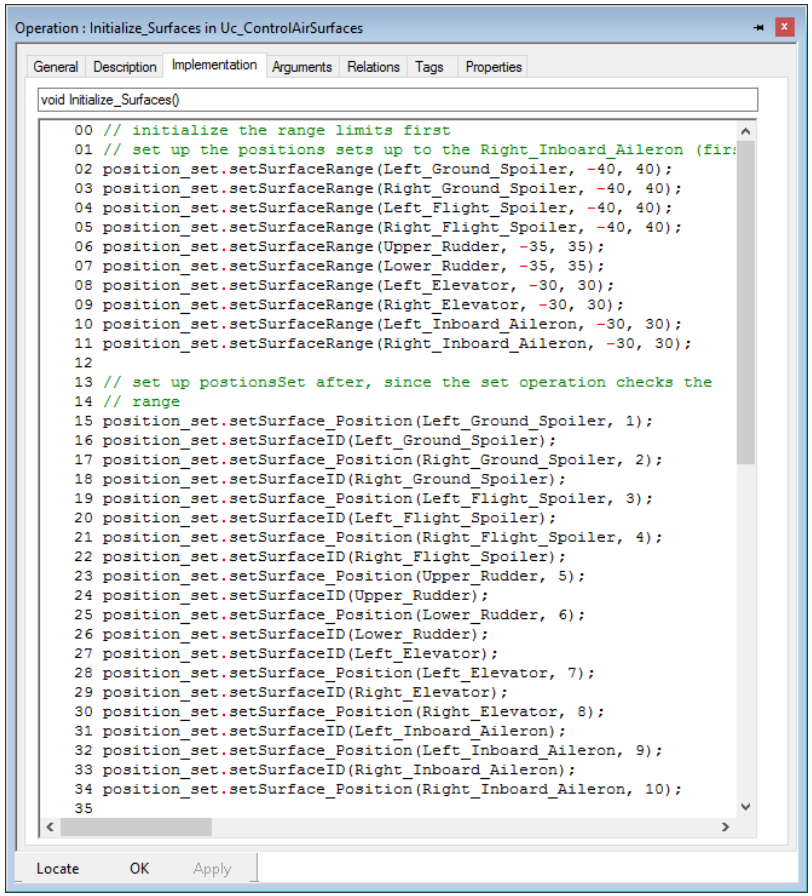
- To **CAS_Surface_Positions**, we'll add an operation to set the surface ranges for individual surfaces



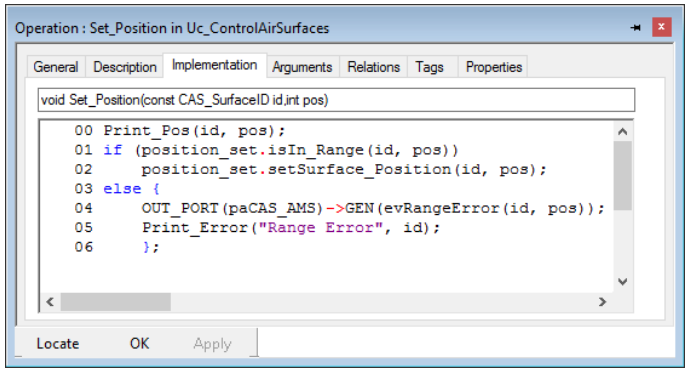
and we'll add an **isInRange(): RhpBoolean** function to see if a commanded position is between the low and high range limit for a particular surface



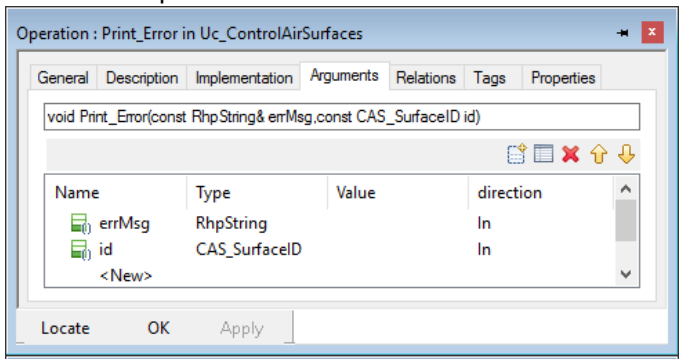
- To the **Uc_ControlAirSurfaces** use case block, we'll add an **Initialize_Surfaces** operation to set the values of the surfaces. This operation will be invoked when we start the state behavior.

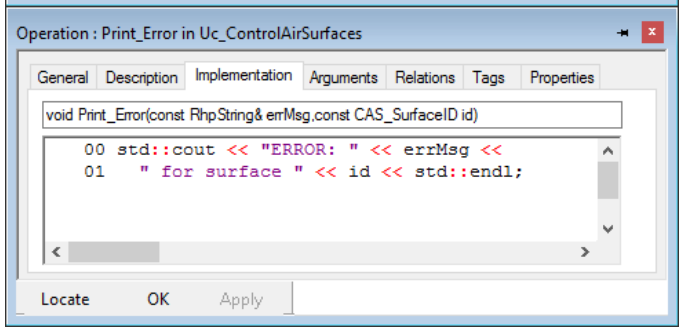


- We'll also update the use case block operation **setPosition** to check the range before assigning the value, and issue an error message if not



You'll notice that this function calls a new operation called **Print_Error** in the use case block as well to send that information to standard output:

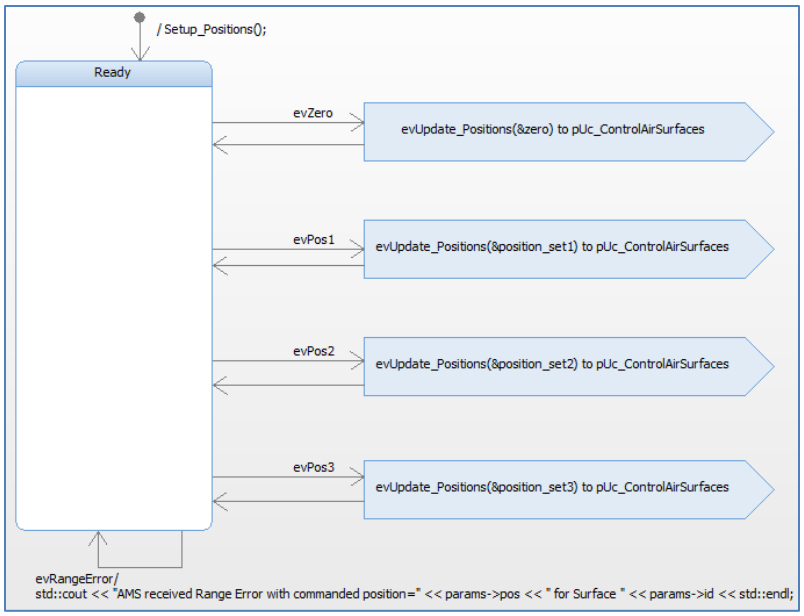




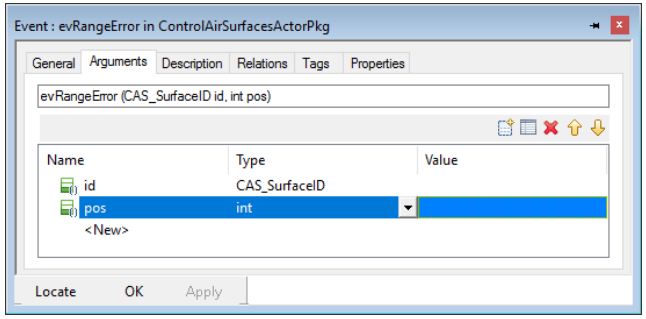
- If you recall, the actor block **aCAS_AMS** is already set up with **zero**, **position_set1**, **position_set2** and **position_set3** surface position sets. All but **position_set3** have in-range values but **position_set3**

has out of range values. We can now run the model and send the events **evPos2** and **evPos3** to the actor to ensure that good values pass and bad values are detected.

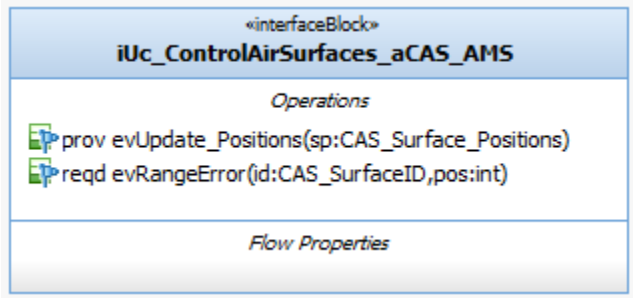
- We must also update the state machine for the **aCAS_AMS** actor block to receive the event **evRangeError**, which passes the surface ID and the commanded value.



- Next, edit the event **evRangeError** to add the parameters (the event will be in the **ControlAirSurfacesActorPkg** package:

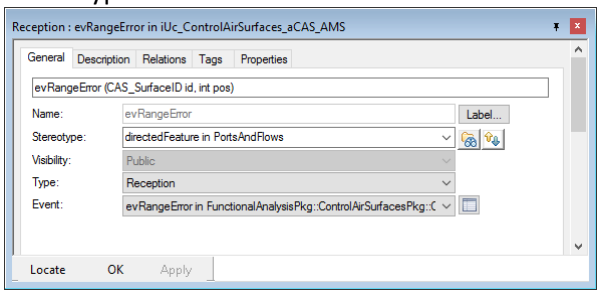


- And, finally, we add the event **evRangeError** to the interface **iUc_ControlAirSurfaces_aCAS_AMS**

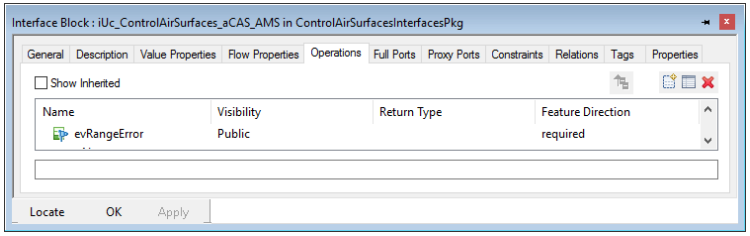


Notice that the **evUpdate_Positions** event is *provided* while the **evRangeError** event is *required*. This is done in a couple of steps

- First, in the *Features* dialog for the event, add the stereotype *directedFeature* to the event.



- Next, in the features dialog for the **iUc_ControlAirSurfaces_CAS_AMS** interface block operations tab, set the direction of the event flow in the *Feature Direction* field.

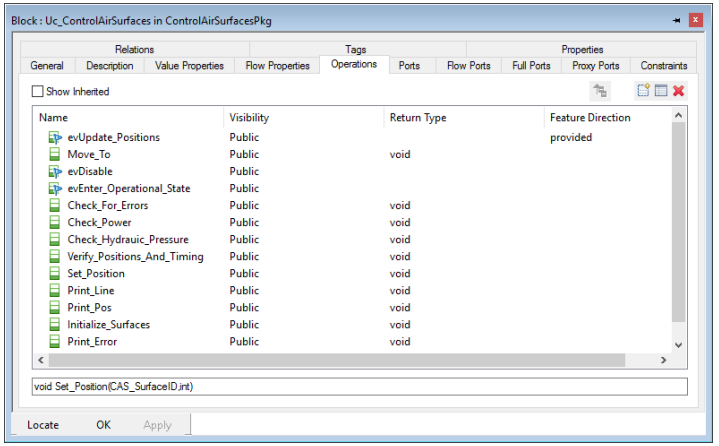


Having trouble getting your objects to communicate?

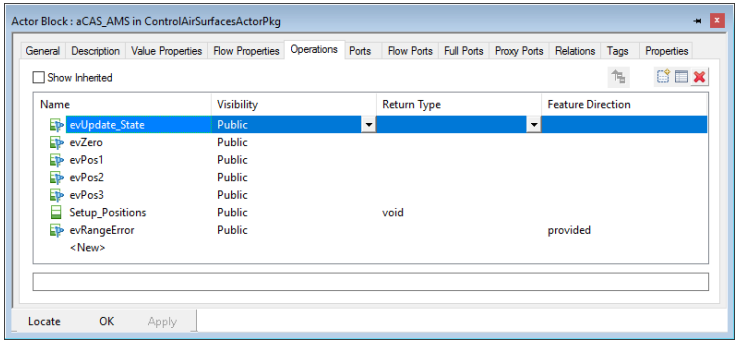
Sometimes, the model compiles and runs but messages sent from one object to another don't seem to arrive. Things to check:

- Is the event in the interface?
- Is the event or call stereotyped as a *directedFeature?*
- Are the ports marked as behavioral?
- Is the event direction *provided* where it will be processed?
- Is the event in the interface block *provided* for the unconjugated end if acted on by that instance?
- Is the event in the interface block *required* for the conjugated end if acted on by that instance?
- Is there link between the ports on the different instances?
- Are you sure you're looking at the right instances? There may be multiple instances of a block.
- Try going to the folder that has the generated code and object files (a subdirectory of your model folder) for the configuration you're using, delete all the code and object files there and completely regenerate.

- In the features dialog for the **Uc_ControlAirSurfaces** use case block, make sure the **evUpdate_Positions** event is *provided*



- Finally, in the actor block **CAS_AMS**, set the direction for the **evRangeError** to be *provided* (remember, it is *required* in the interface, and this is the conjugated end of the connection).

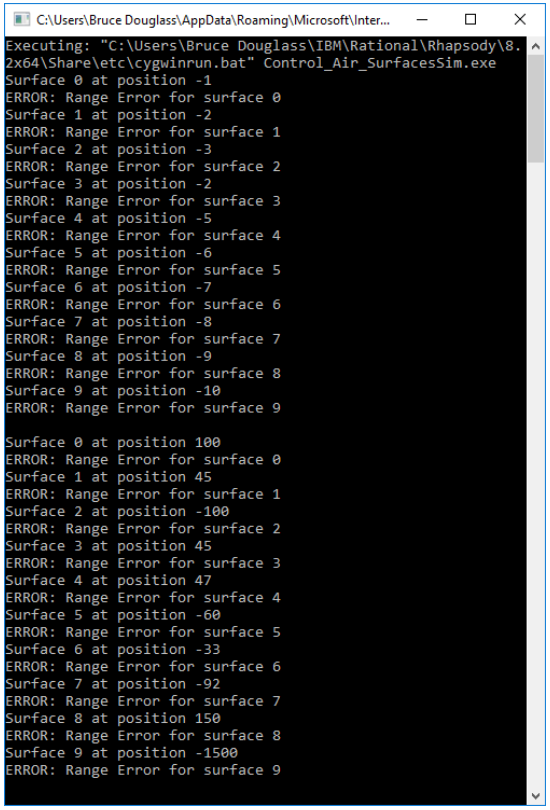


We won't model the movement of the surface itself since requirements are not about *how* something is achieved but rather *what* must be achieved. For our purpose, it is enough to ensure results that are in or out of range

and ensure that the externally visible behavior (such as reporting errors) meets both our needs and the requirements.

We'll need to add some behavior to set up these configuration values for the purpose of simulation support even though that behavior is actually part of the **Configure System** use case. Because we are adding it just to support simulation, we'll label it with the **«nonNormative»** stereotype from the HarmonySE profile to indicate that this isn't a requirement here but is just here to facilitate the simulation.

Now we can run the simulation. Let's send **evPos2** followed by **evPos3** to the actor block. The standard output window should look like this



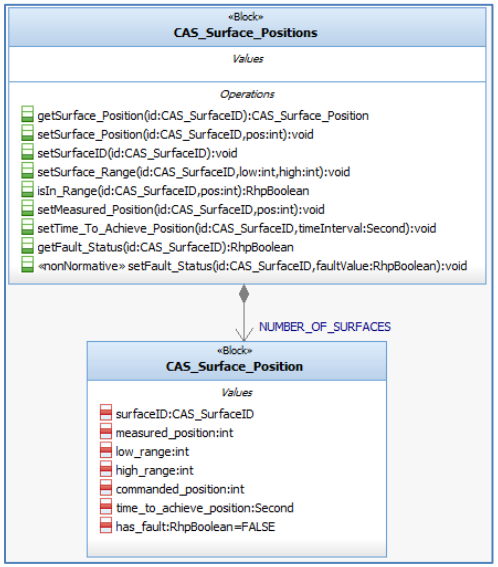
You can see that the first set of values worked fine, while the second set resulted in errors, just as expected.

Step 3: Ensure Accuracy and Timing

Remember that we're not really interested in moving the surfaces here. We are focused on ensuring that the requirements are complete, consistent, accurate, and correct (and capturing the logical interfaces). Requirements focus on externally visible aspects of the system such as when behavior works correctly or incorrectly that this results in proper externally visible outcomes. For example, we have requirements about the accuracy and timing of position movement and if these are violated, the system is expected to notify the **AMS** actor of this fault. That interaction should be captured in our requirements model even though we're not actually designing the movement of the surfaces.

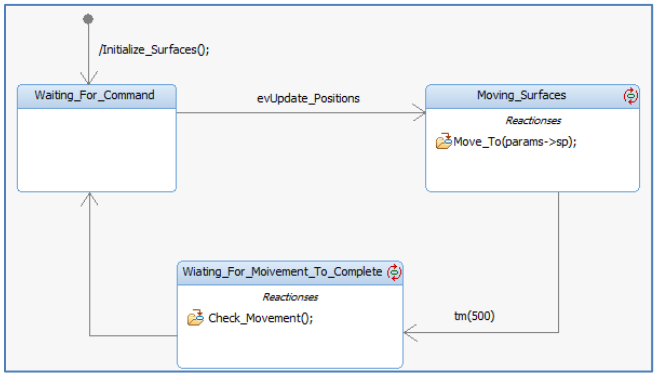
To simulate this, we need to add structure and behavior to the use case block to represent the measured position and the time required to complete the movement. Those properties will need to be added to the **CAS_Surface_Position** block. We'll also need to add an operation to the **CAS_Surface_Positions** block to get that information from each surface.

You can see that we've added a **measured_position** value. This will simulate the position actually achieved. The **commanded_position** value holds the commanded position.



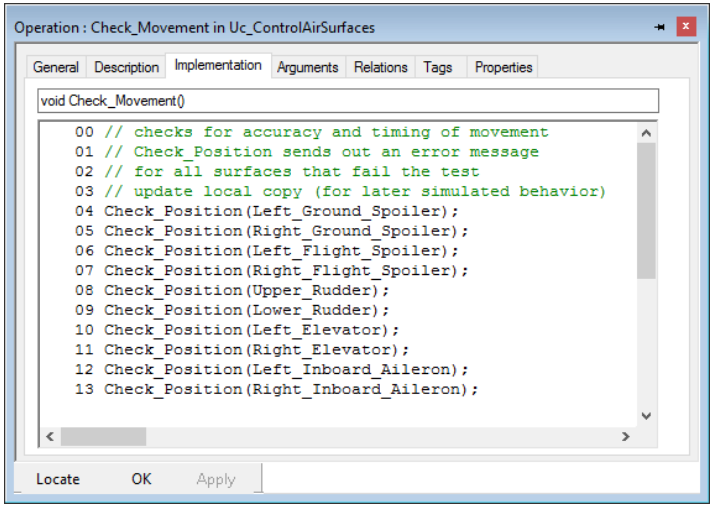
Note that we used the *Second* unit type from the SysML profile model library. To set the value property to be of this type, in the features dialog for the value property **time_to_achieve_position**, in the *Type* drop down list, click on Select and navigate to *Profiles > SysML > SIDDefinitions > BaseSIUnits*.

These movements may take some time, so we'll modify the use case block state machine to check the positions once they're done.

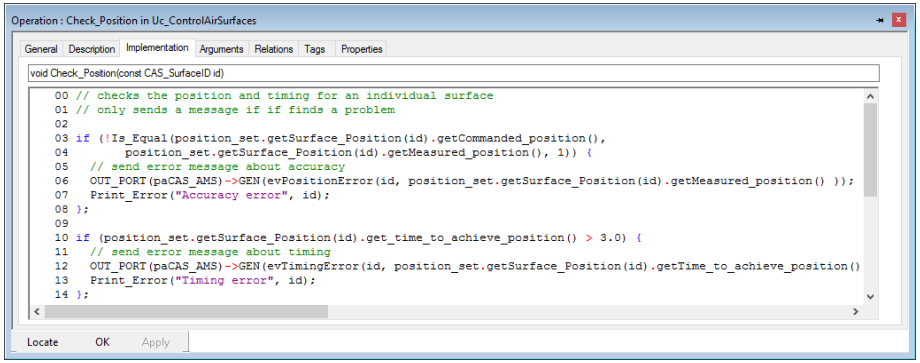


I used a timeout to drive the transition although arguably it should be driven by the completion of the movement. However, I'm not trying to design the internal system functionality, but rather to provide the appearance of doing so to the actors. The timeout is a very simple means to disconnect setting the positions from subsequently checking the outcomes.

The normal behavior of **Set_Positions** sets the value **commanded_position** of the surface (and really, would normally set **measured_position** to the same value). The **Check_Movement** operation must check the acquired position against the commanded position as well as check the timing of the movement.



We also need to add the **Check_Position** operation to the **Uc_ControlAirSurfaces** use case block. This takes a single parameter, the surface id:



The implementation – if too small to read – is:

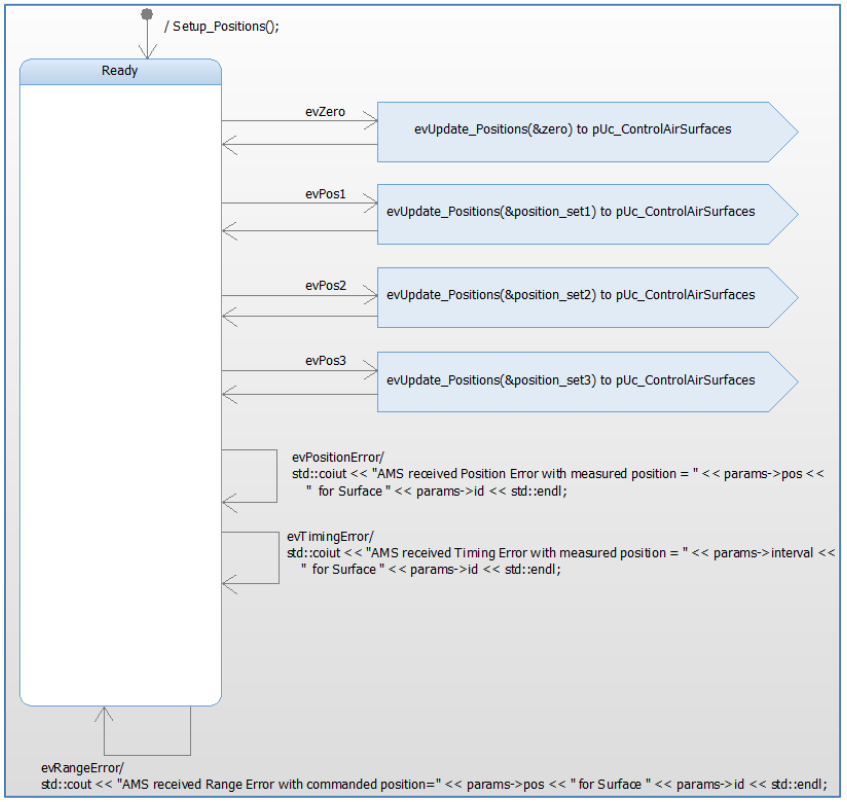
```
// checks the position and timing for an individual surface
// only sends a message if if finds a problem

if (!Is_Equal(position_set.getSurface_Position(id).getCommanded_position(),
       position_set.getSurface_Position(id).getMeasured_position(), 1)) {
  // send error message about accuracy
  OUT_PORT(paCAS_AMS)->GEN(evPositionError(id,
position_set.getSurface_Position(id).getMeasured_position() ));
  Print_Error("Accuracy error", id);
};

if (position_set.getSurface_Position(id).get_time_to_achieve_position() > 3.0) {
  // send error message about timing
  OUT_PORT(paCAS_AMS)->GEN(evTimingError(id,
position_set.getSurface_Position(id).getTime_to_achieve_position()));
  Print_Error("Timing error", id);
};
```

To compare two values, let's add an **Is_Equal** operation to the use case block that accepts three *RhpReal* parameters (**a**, **b**, and **tolerance**), and returns **TRUE** if the difference between the first two values is less than the tolerance:

```
return abs (a-b) <= tolerance;
```

So if the tolerance is, say 1 and we have a commanded position of 18 and a measured position of 19, the values would be said to be equal.

We must also update the **aCAS_AMS** actor block state machine to receive the **evPositionError** and **evTimingError** events:

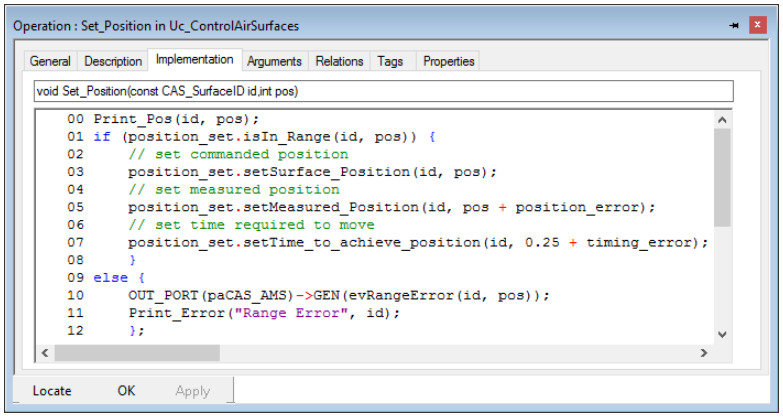


The **Check_Movement** operation will also have to check the timing. The attribute **time_to_achieve_position** will be generally be set to a passing value, but we want the **ability** to set it to a failing value when necessary.

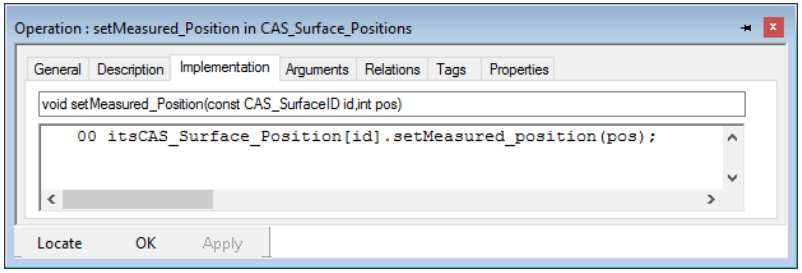
The last thing we must do is set the values of **measured_position** and **time_to_achieve_position** for the surfaces. We'll do this by modifying the **Uc_ControlAirSurfaces** use case block. Nominally, we'll just set the measured position to be the same as the commanded position and the time to a short value, such as 0.25s. We also want to generate position and timing errors, so we'll add new values in the use case block: **position_error** (of type *int*) and **timing_error** (of type *Second* or *double*). We'll modify **Set_Position** to add these values to **measured_position** and **time_to_achieve_position**, respectively. When we're running, we can

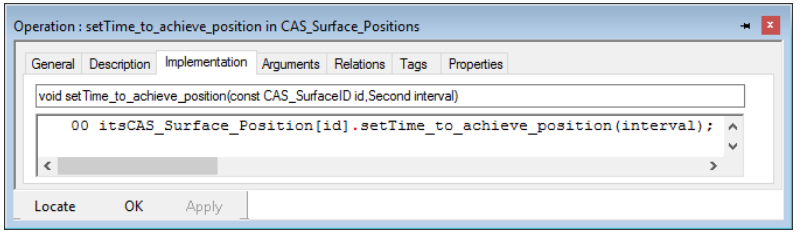
change these values and thereby simulate error conditions. These attributes are only here to support the simulation per se and not represent requirements. Therefore they are non-normative and are stereotyped as such.

The updated **Uc_ControlAirSurfaces::Set_Position** operation now looks like this:

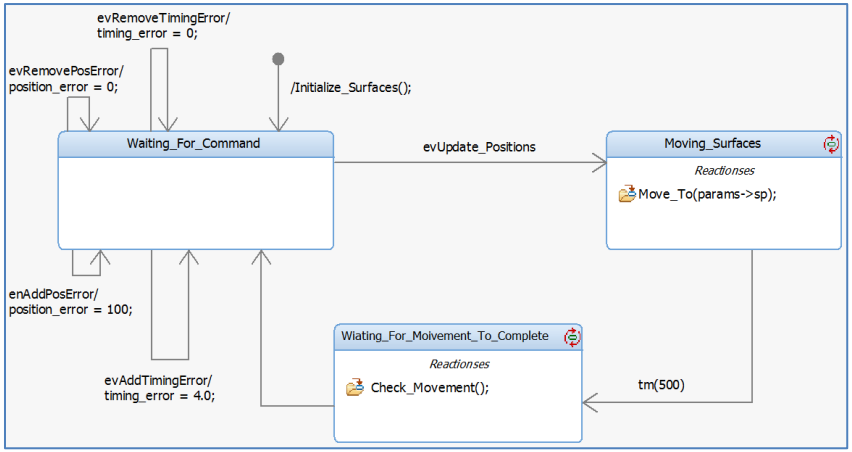


We also add **setMeasured_Position** and **setTime_to_achieve_position** operations to the **CAS_SurfacePositions** block (note the argument lists):



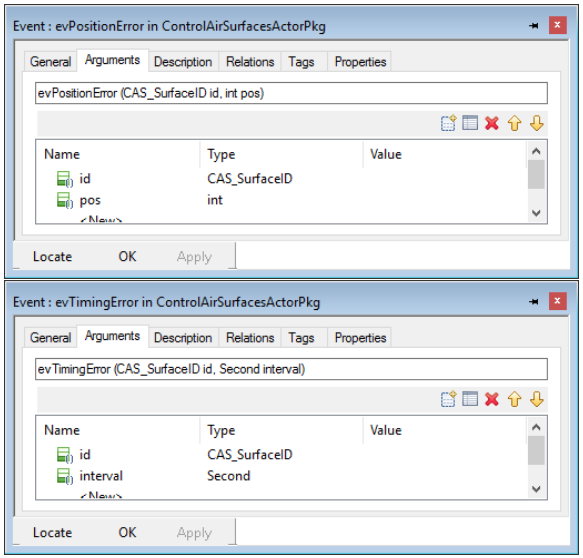


We can modify the values of attributes as we run the model, but let's add events to the use case block state machine to set both position and timing errors to make the simulation a little easier:

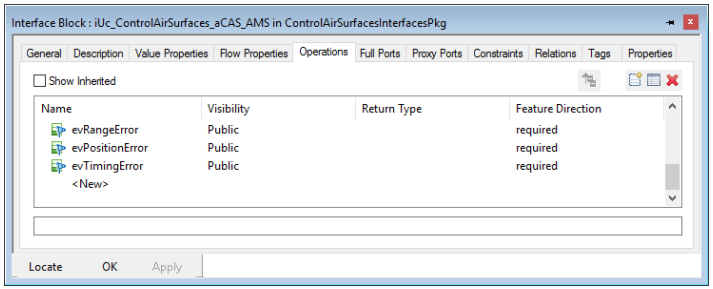


The event used above to set and remove error conditions should be sterepotyped as nonNormative.

You must add the arguments to the events **evPositionError** and **evTimingError**.



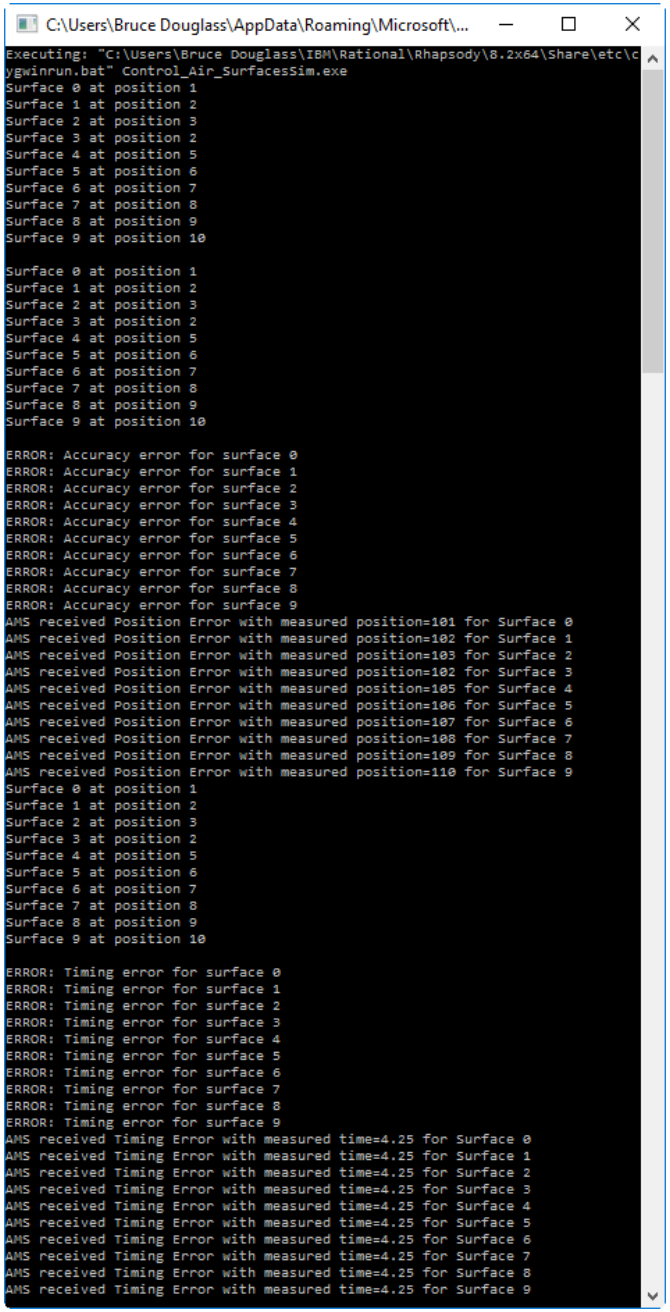
Don't forget to add the events **evPositionError** and **evTimingError** to the interface **iUc_ControlAirSurfaces_aCAS_AMS** and make them directed features (direction: *required*), as we did for the **evRangeError** event (not all event receptions are shown):



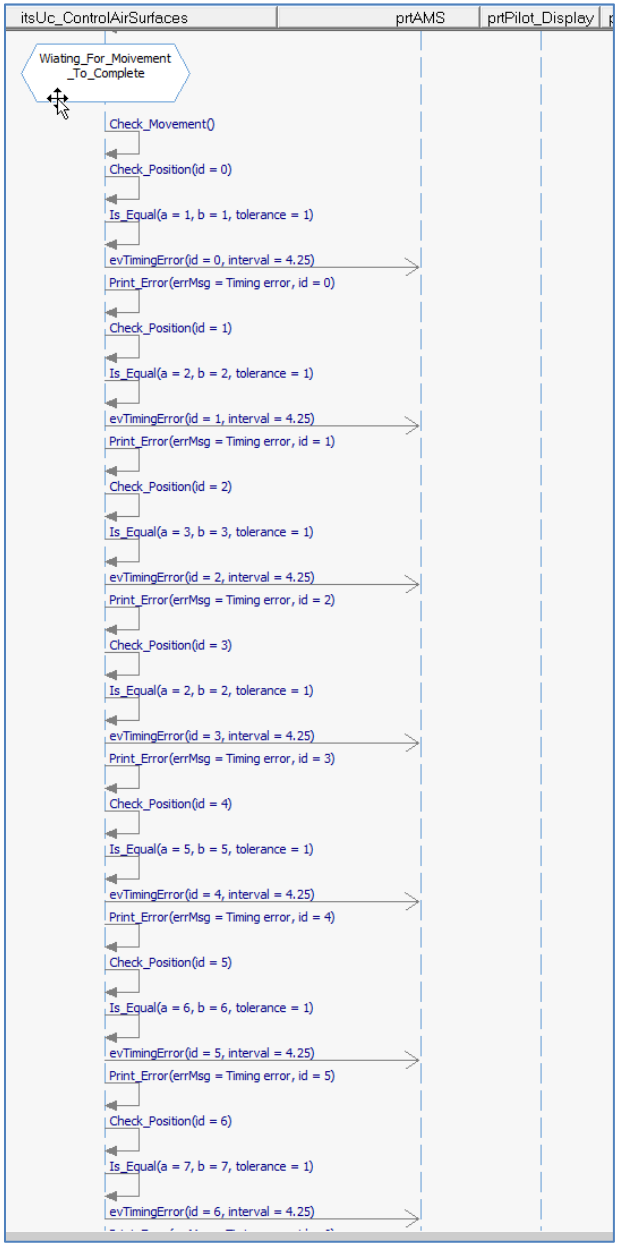
Let's now compile and run the simulation with the following case:

- Actor block **pos1** sent (by sending the **evPos1** event to the actor block instance)
- set position error and send actor block **pos1**
- Reset position error, add timing error, and send actor block **pos1**

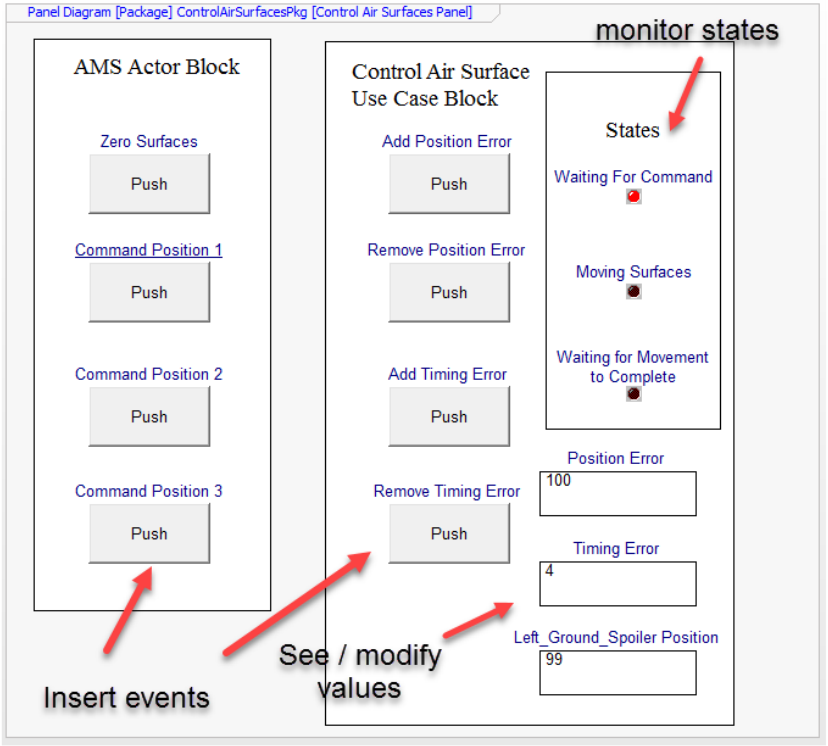
The sequence diagram for that interaction is quite long, so here is the text send to standard output:



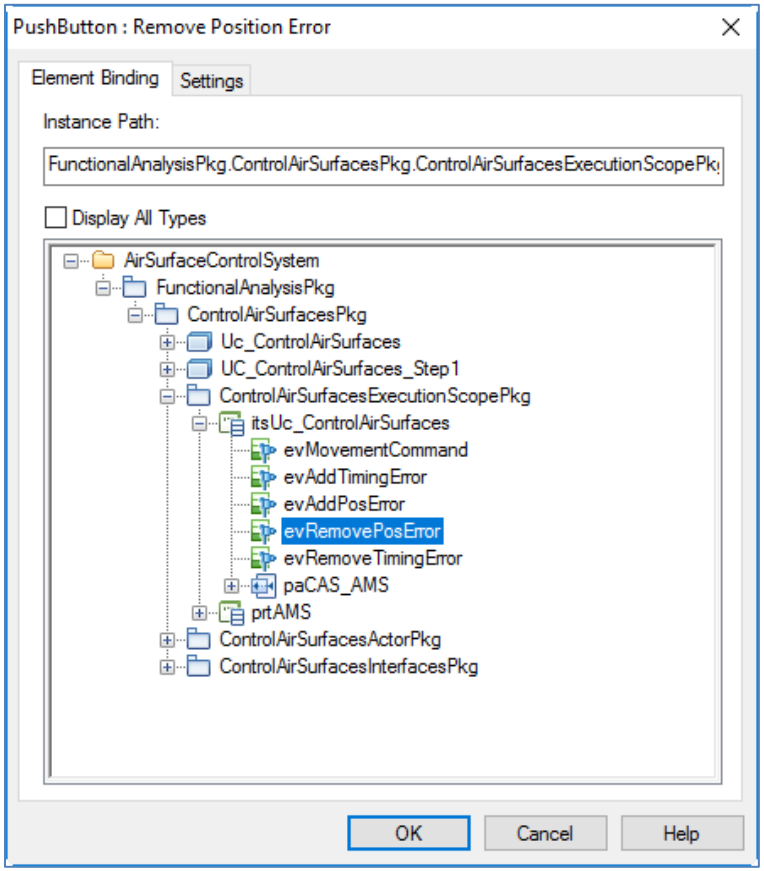
Here's what part of the sequence diagram looks like for the case when position errors are created, beginning with the call to **Check_Movement**:



If you've been entering the events using the event generation tool, you can also do with via a panel diagram. In this case, create a panel diagram and add push buttons as shown in the diagram:



These push buttons are bound to the event receptors of the block instances. If you want to create this diagram to assist in driving the simulation, be sure to select the instances in *the FunctionalAnalysisPkg > ControlAirSurfacesPkg > ControlAirSurfacesExecutionScopePkg > Parts* area of the model. You can get there by double clicking on the push button and navigating to the desired part and selecting the event reception.



Step 4: Handling requirements about warm and cold restarts

There are some requirements about warm and cold restarts such as

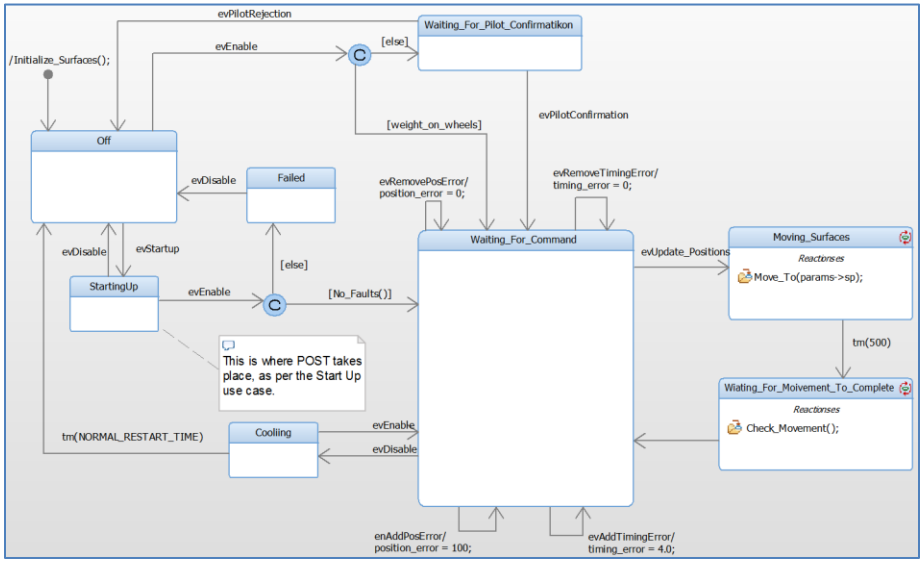
The system shall not automatically perform minimum, maximum, and zero position tests during a restart, where "restart" is defined to be starting up within 5 minutes after being enabled, or being operational. Rationale: this is to allow in-flight restarts safely.

and

The system can be commanded into a restart mode from the OFF_STATE by the Attitude Management System. In addition, the system may be explicitly commanded into restart from other operational modes with an independent command from the AMS, however, this command must be verified by soliciting and receiving a pilot override instruction. An exception to this is if the plane is not Weight on Wheels (WoW); in this case, the restart shall not require an independent pilot confirmation.

In this next nanocycle, let's add this behavior. Note that the actual execution of the start tests is the subject of the (previously analyzed) **Start Up** use case. For that reason, the tests will not be modelled here. What will be modelled is a placeholder for them. That placeholder is an example of a small, but important, overlap between use cases.

Here is the updated state machine for the **Uc_ControlAirSurfaces** use case block:

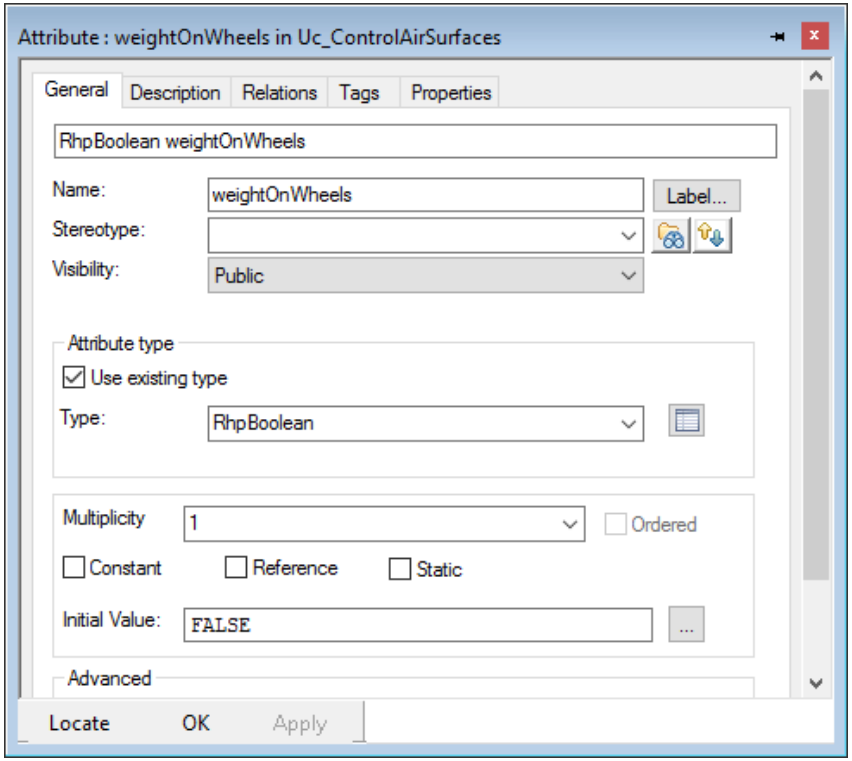


We'll need to provide some behavior for the **No_Faults** and **Wow** operations used in the state machine, and defined within the use case block. We'll also define the constant **NORMAL_RESTART_INTERVAL** , which is nominally 5 minutes (we'll set it to a shorter value, such as 10s for the purpose of simulation). In the **ControlAirSurfacePkg > ControlAirSurfacesTypesPkg**, add the following type by right clicking on the package and selecting *Add New -> Blocks > DataType*. Name this type **NORMAL_RESTART_INTERVAL**. Then double click on it, ensure that the *Kind* is *Language*, and its declaration to be

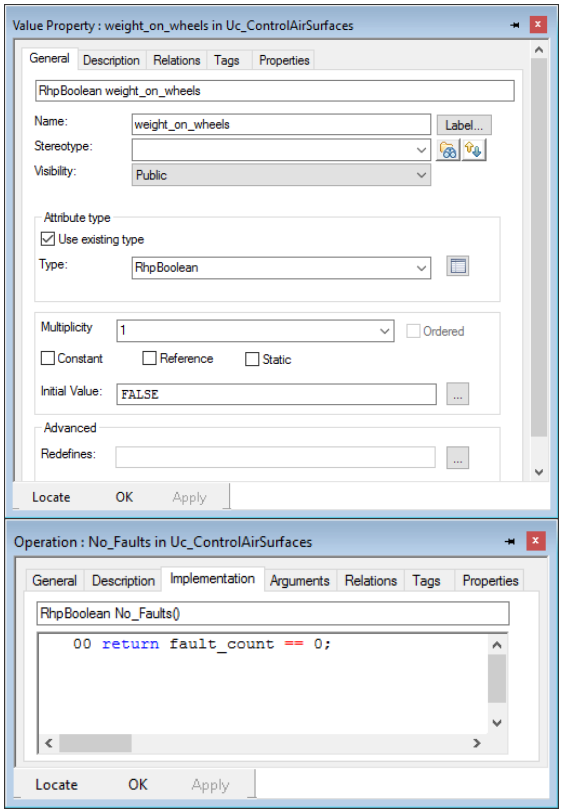
```
#define %s 10000
```

This will give the timeout on the state machine a 10 second interval, suitable for simulation.

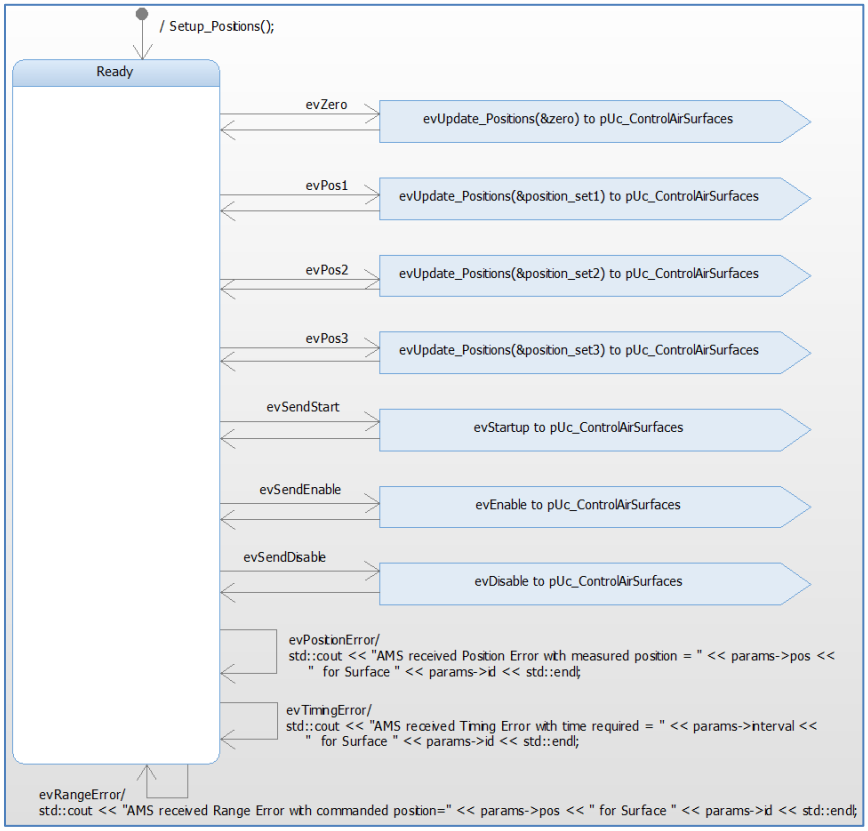
We'll define a **RhpBoolen** type attribute in the use case block named **weightOnWheels** and give it a default value of **FALSE**. (We can change it later during simulation if desired).



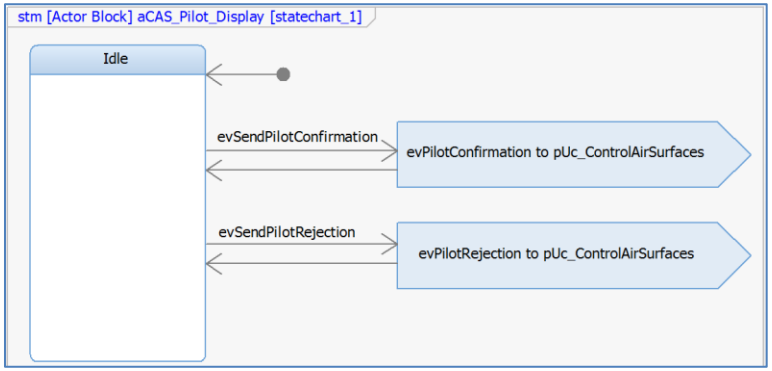
We'll do the same kind of thing for the **No_Faults** operation. Define an attribute for the use case block named **fault_count** of type *int* and then define the function **No_Faults** to return *TRUE* if that value is zero.



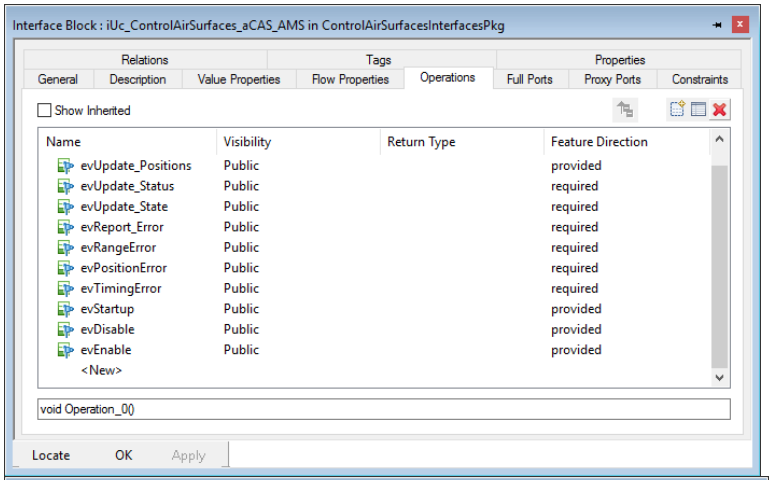
The corresponding updated **aCAS_AMS** state machine must be able to generate the **evStartUp**, **evEnable**, and **evDisable** events.

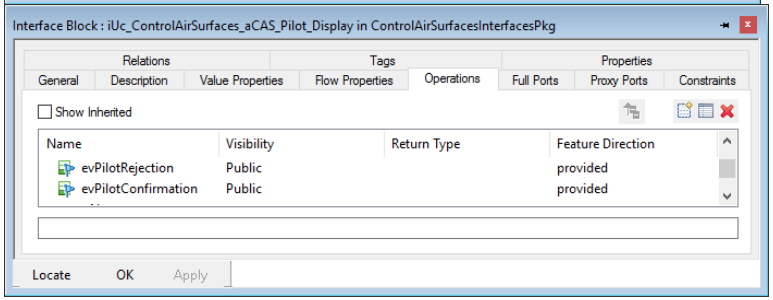


We also need to add the ability of the **aCAS_PilotDisplay** to generate the **evPilotConfirmation** and **evPilotRejection** events:

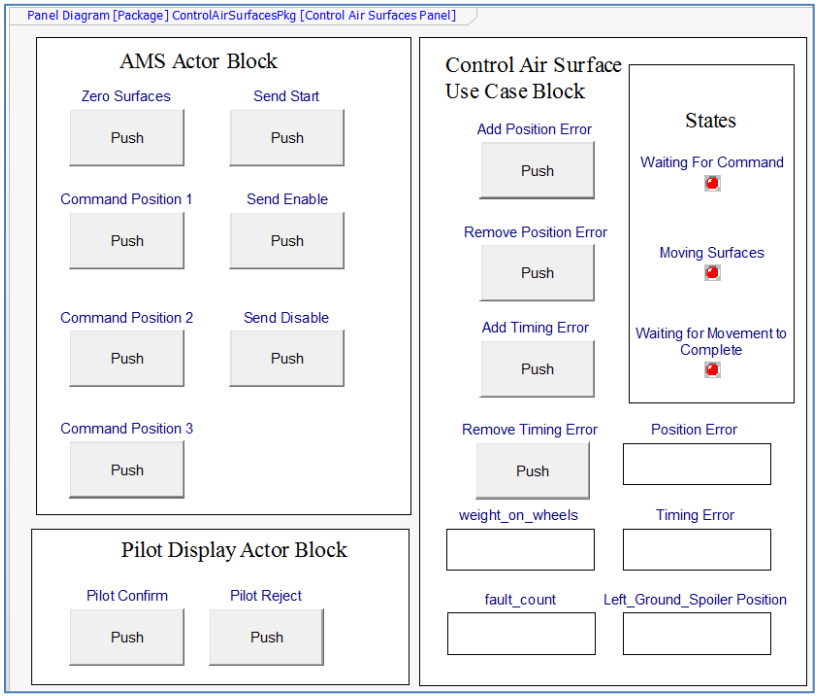


We will need to add the **evStartUp**, **evEnable**, and **evDisable** receptions to the use case block and the interface as provided *directedFeatures*. We must also add the **evPilotConfirmation** and evPilotRejection events to the **iUc_ControlAirSurfaces_aCAS_Pilot_Display** interface block in the **ControlAirSurfacesPkg**, again as a provided *directedFeature*. Be sure to add the *directedFeature* stereotype to the event reception in the use case block as well.





Finally, we'll update the panel diagram to help use drive the simulation.



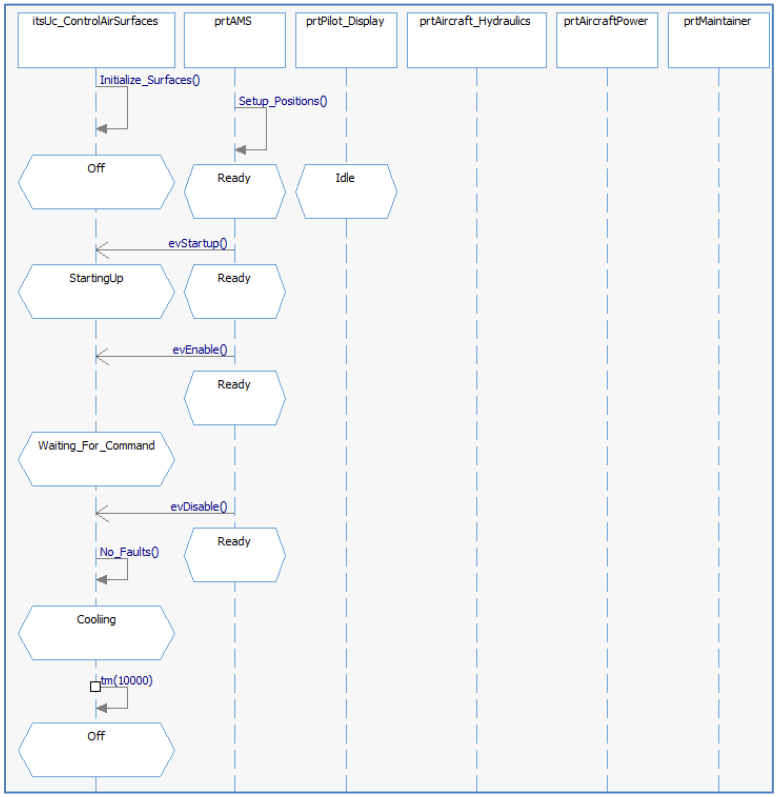
The **weight_on_wheels, position_error, timing_error** and **fault_count** value properties are bound to the text box controls on the panel diagram.

We've added some interesting flows here, such as

1. Normal flow to get to operational mode (**Waiting_For_Command** state) with no errors, then back of the off state.
2. Error flow where we go to Failed state because power on self tests failed.
3. Warm restart within the warm restart interval
4. Directly running from **Off** with Pilot Confirmation
5. Directly running from **Off** with weight on wheels (aircraft on the ground)

We'll look at a few of these. You are encouraged to execute remainder of them to fully explore the requirements.

Here's the simulation run of the first flow:



Want to see the states in your sequence diagrams?

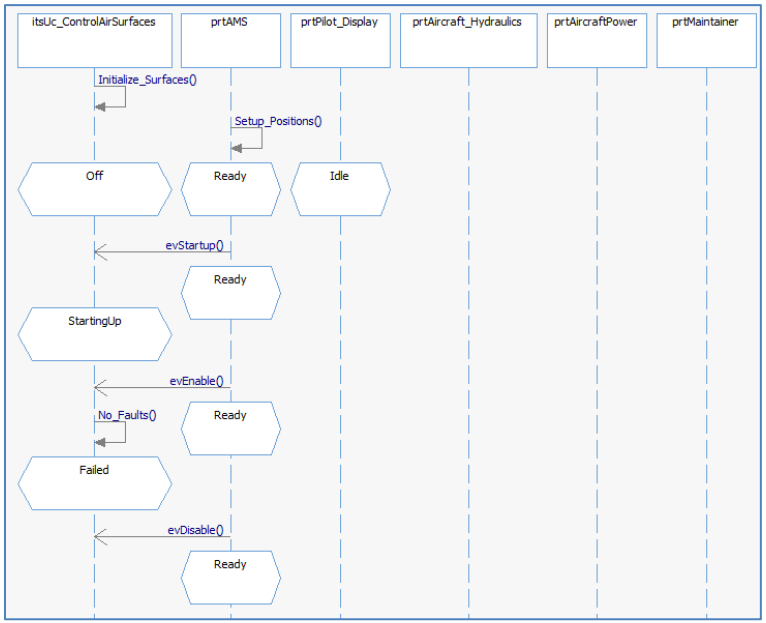
Double click on the project, select the *Properties* tab, select *View All*, then set the *SequenceDiagram > Systems Modeling (tab) > ModShowAnimStateMark* checkbox.

You can see that the 10s timeout occurred after we entered the **Cooling** state. So the model run as expected.

Want to see to see horizontal messages in your sequence diagrams?

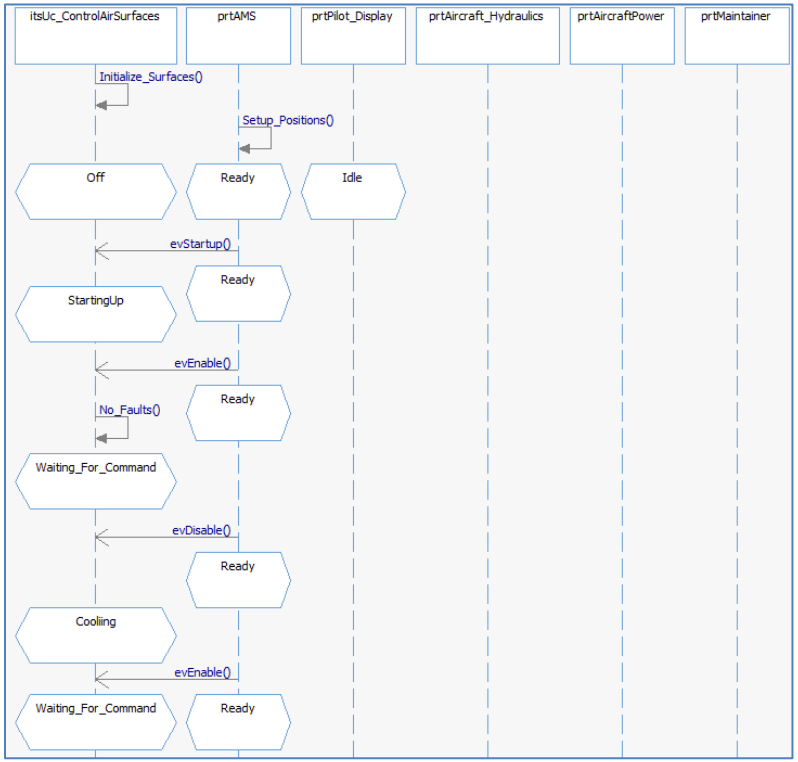
You may notice that asynchronous events are displayed as angled lines. This is because they show when the events were actually send and received. This can make the sequence diagrams less readable. You can fix this by saving the animated sequence diagram (trying to close it will result in a popup asking you if you want to save the diagram). Then reopen the diagram, right click in the diagram and select *SE-Tookit > Straighten Messages*.

The second flow is when the power on self tests (POST) fail; in this case, control should proceed to the **Failed** state. To execute this flow, run the model and use the panel diagram to set the value of **fault_count** to a non-zero value (such as 3). Then send the **evStart** event followed by the **evEnable** event.

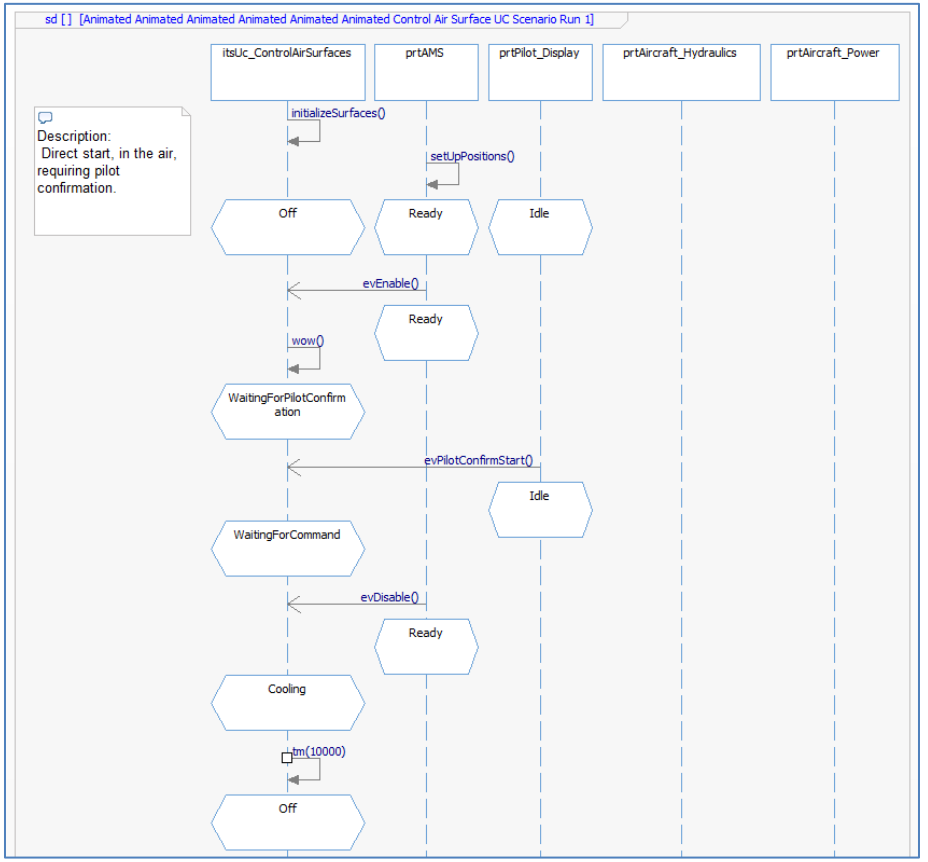


Trouble setting values with the Panel Diagram?
When using text box to set values on the Panel diagram, Rhapsody calls the mutator operation for the attribute. If code generation for attributes uses *smart* generation, then these operations are sometimes not be created. You can force them to be created by double clicking on the project, going to the *Properties* tab, selecting *View All*, then setting *CPP_CG > Attribute > MutatorGenerate* to *Always*.

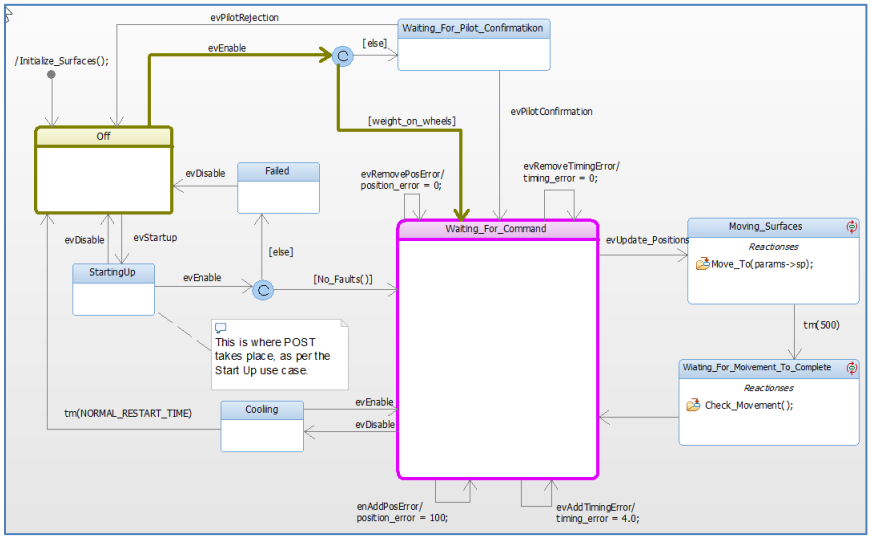
Flow 3 is generated by going through the normal start up sequence (**evStartUp** followed by **evEnable** with **faultCount** set to 0), sending an **evDisable** event followed by an **evEnable** event in less than the **NORMAL_RESTART_INTERVAL**.



The fourth flow entails trying to go directly to operational mode while on the ground, and thus not requiring pilot confirmation. After starting the simulation, use the panel diagram to ensure that the value of **weight_on_wheels** is 0 (FALSE) and then send the **evStart** event.



Scenario 5 is shown below in the state machine with the last state, then last transition path, and the currernt state hightlighed. In this case, the model is run and the value of **weight_on_wheels** is set before the **evEnable** event is sent. Since the value of **weight_on_wheels** is *TRUE*, the **Waiting_For_Command** state is achieved.



Step 5: Manage "flyable" operational state with surface faults

This last nanocycle step for this use case analysis adds in the requirements for determining if the system is flyable with one or more surface faults.

As we start to analyze this, we discover that what is considered a "Flyable set" of surfaces isn't identified. These are missing requirements. As systems engineers, we need to talk with the subject matter experts of the airframe to discover those requirements. For the purpose of discussion, they responded to our solicitation with the following three new requirements, which we will enter into our model and then allocate to the use case:

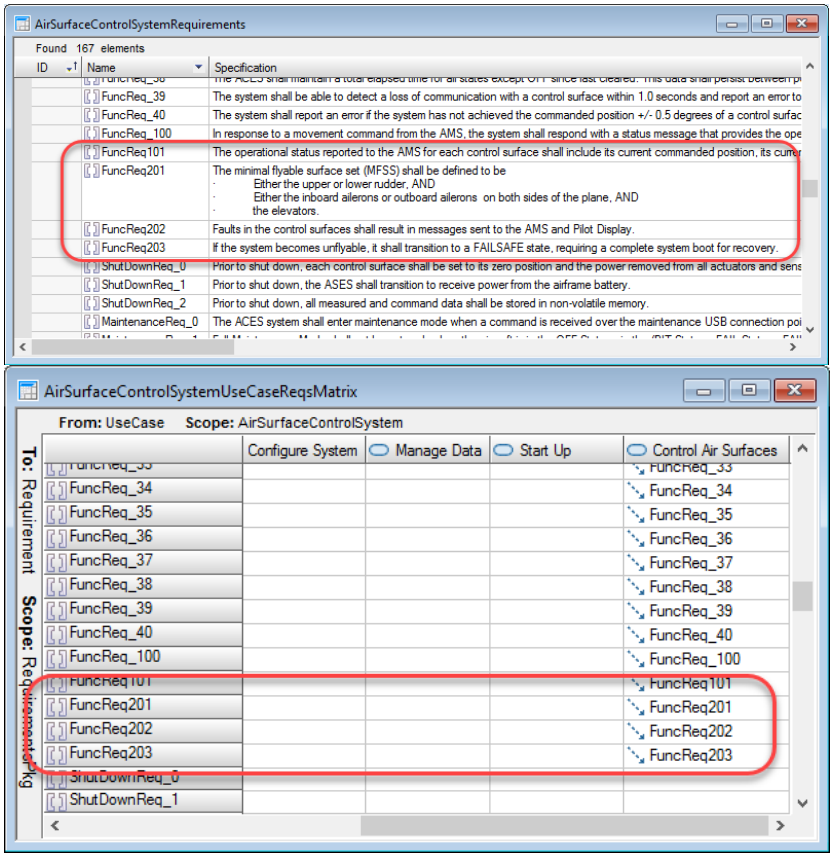
The minimal flyable surface set (MFSS) shall be defined to be

- *Either the upper or lower rudder, AND*
- *Either the inboard ailerons or outboard ailerons on both sides of the plane, AND*
- *the elevators.*

Faults in the control surfaces shall result in messages sent to the AMS and Pilot Display.

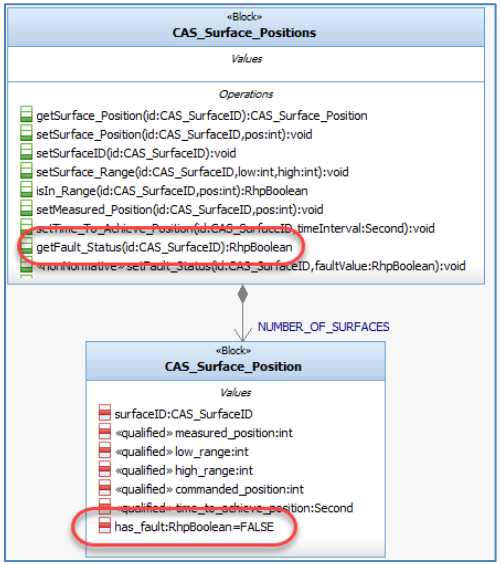
If the system becomes unflyable, it shall transition to a FAILSAFE state, requiring a complete system boot for recovery.

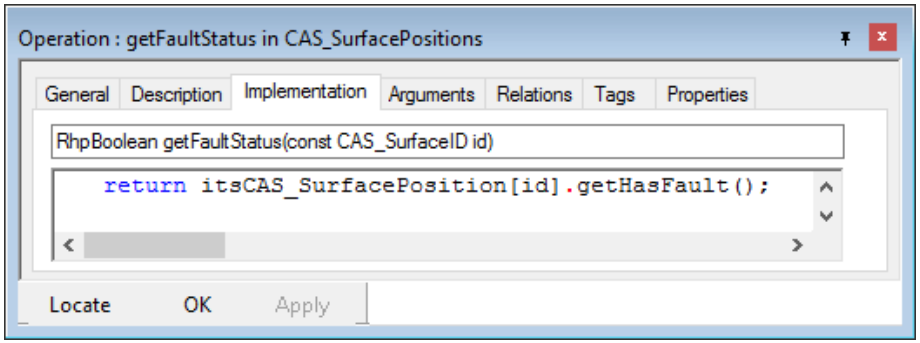
We'll need to update the **Uc_ControlAirSurface** use case block to be able to identify and evaluate problems using the criteria specified and add tracea from the use case to those new requirements.



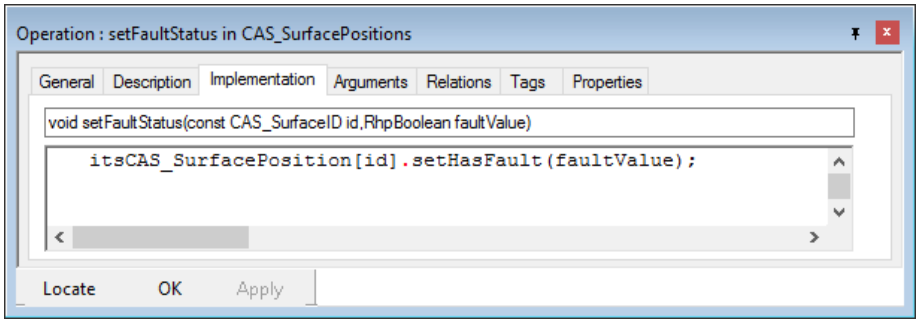
To model the faults, we'll add a fault condition to each control surface; that is we'll add a **has_fault** value property (type **RpyBoolean,** default value *FALSE*) to the **CAS_Surface_Position** block, and a **getFault_Status** operation to the **CAS_SurfacePositions** block to easily get the fault status of any

surface. These blocks are, as you no doubt remember, located in the **ControlAirSurfacesTypesPkg** package.

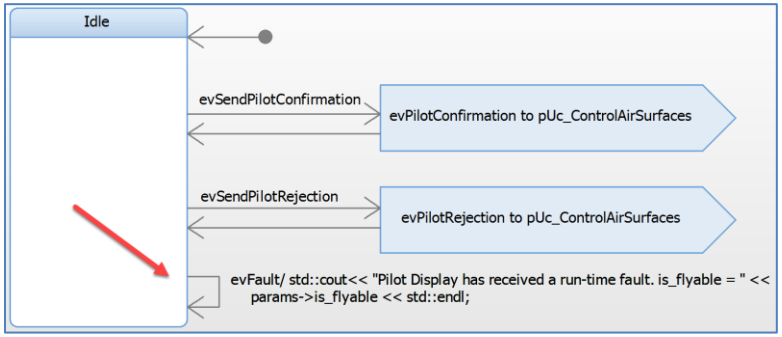


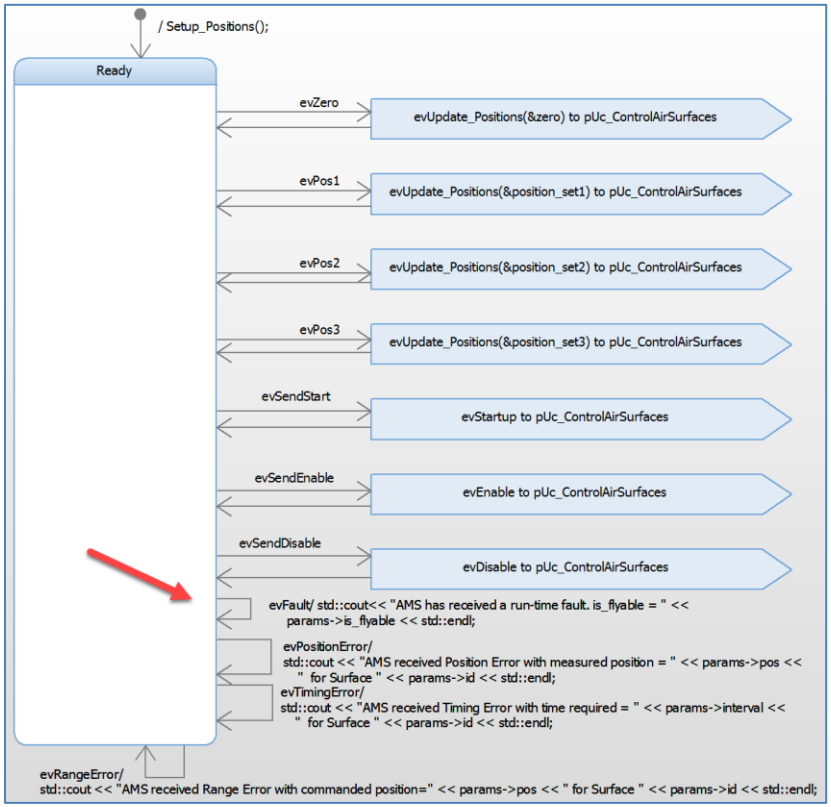


For simulation purposes, we'll also add a non-normative function to the **CAS_SurfacePositions** block to set the fault status of any surface, called **setFaultStatus**. It will take two parameters, an **id** (of type **CAS_SurfaceID**) and a **faultValue** (of type **RpyBoolean**).

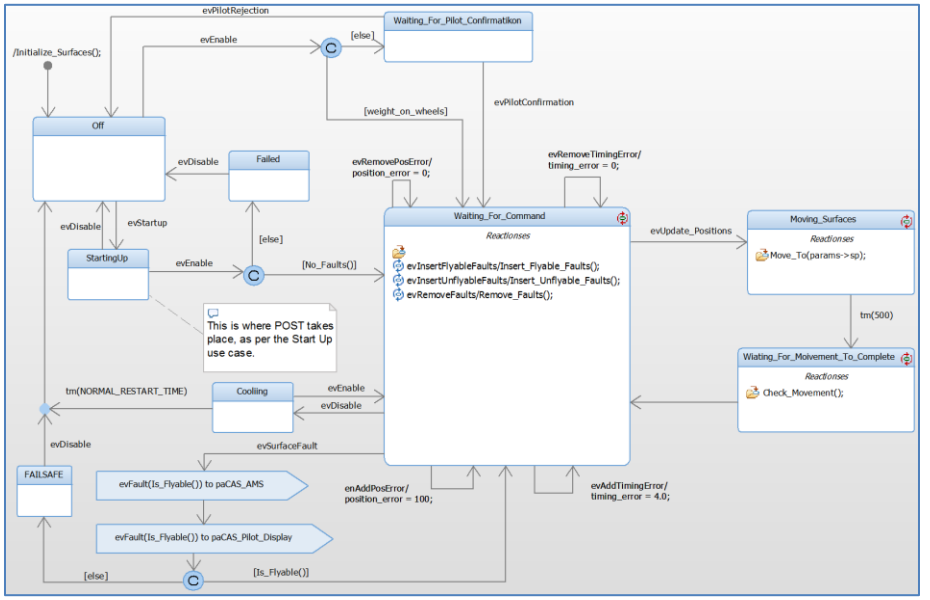


Next, let's update the actors to receive the **evFault** event. Note that the after adding the event to the actors, you'll have to edit the event in the browser to add the **is_flyable** (of type *RHPBoolean*) argument to the event.

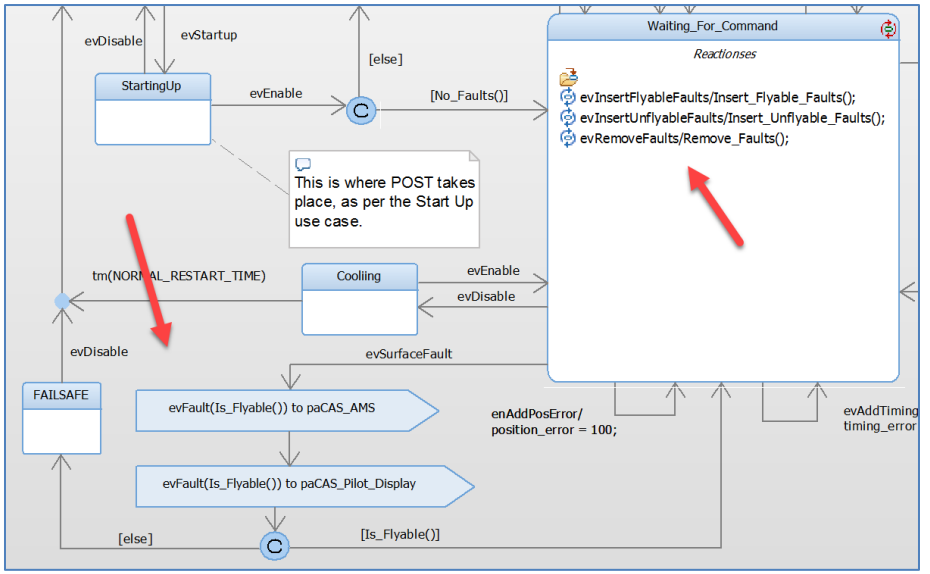




Now, let's update the state machine for **Uc_ControlAirSurfaces** to add the behavior to manage these faults and send the evFault event.



The additions to the state machine are concentrated in the bottom left-hand corner:

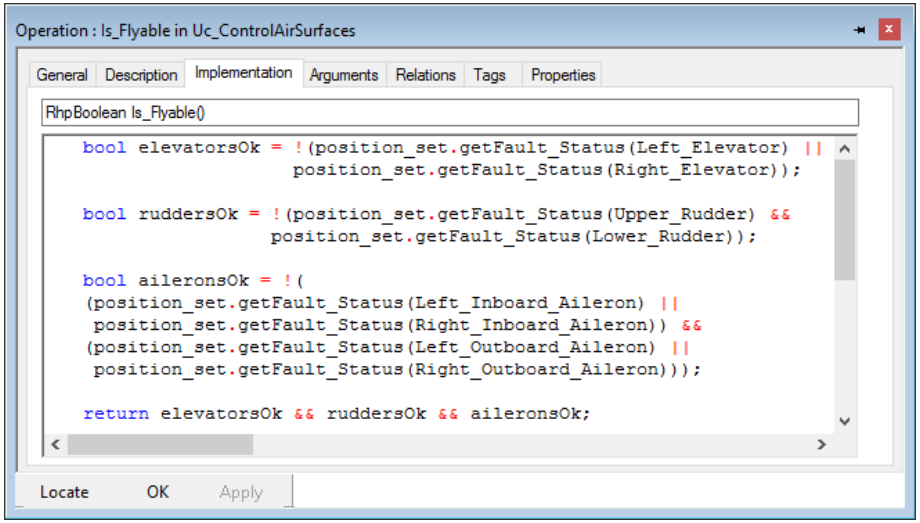


So, if the system is in the **Waiting_For_Command** state and it receives an internal **evSurfaceFault** event, it checks if the remaining surfaces are a flyable set with a call to **Is_Flyable()** to see if the remaining surfaces are in the flyable set. If the guard

```
[Is_Flyable()]
```

returns **FALSE**, then the system proceeds to the **FAILSAFE** state. From there, the system only accepts the **evDisable** event to enter into the **Off** state. If the guard returns **true**, then the system transitions back to the **Waiting_For_Command s**tate.

The implementation of the **Is_Flyable** operation basically checks the entire set of surfaces to ensure that a flyable set is still operations.

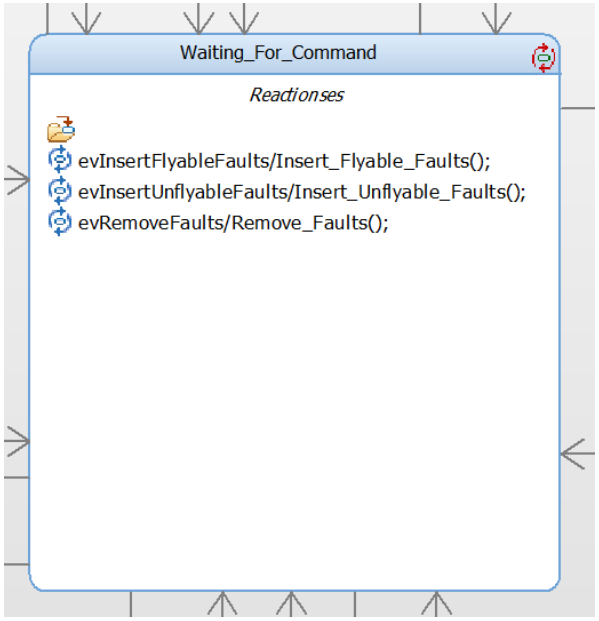


The contents of the implementation field are shown below to make them a bit easier to read:

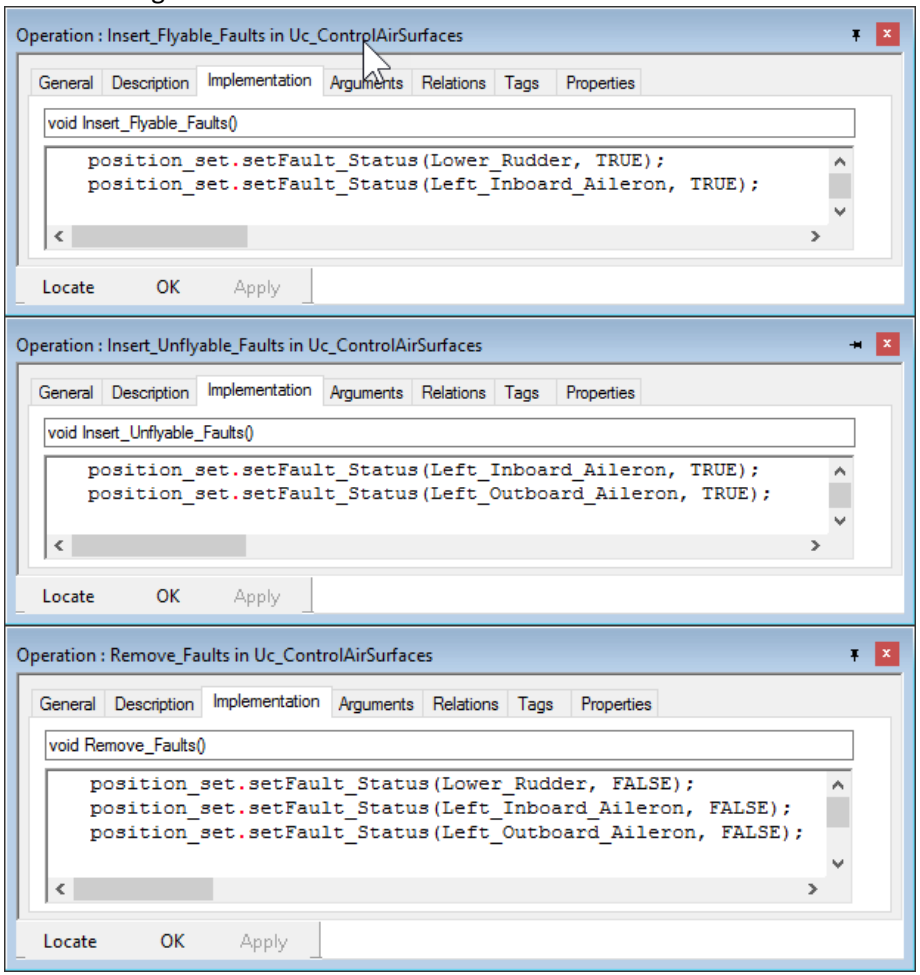
```
bool elevatorsOk =
!(position_set.getFault_Status(Left_Elevator) ||
position_set.getFault_Status(Right_Elevator));

bool ruddersOk = !(position_set.getFault_Status(Upper_Rudder)
&&
                  position_set.getFault_Status(Lower_Rudder));

bool aileronsOk = !(
(position_set.getFault_Status(Left_Inboard_Aileron) ||
 position_set.getFault_Status(Right_Inboard_Aileron)) &&
(position_set.getFault_Status(Left_Outboard_Aileron) ||
 position_set.getFault_Status(Right_Outboard_Aileron)));

return elevatorsOk && ruddersOk && aileronsOk;
```

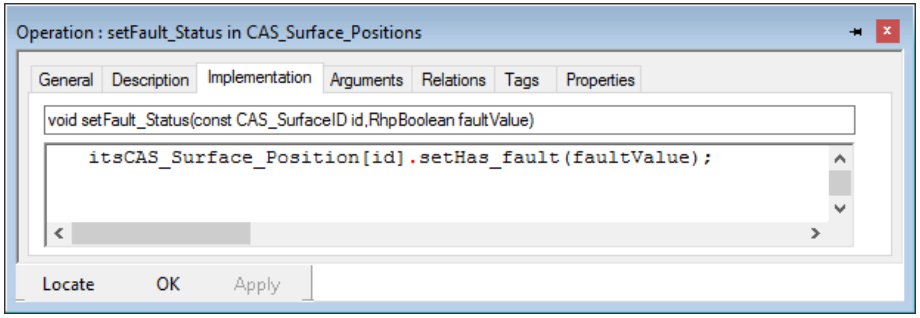
Especially note the **not** operators (“!”) in the code.

Let’s add some internal transitions to the **WaitingForCommand** state to add and remove faults:



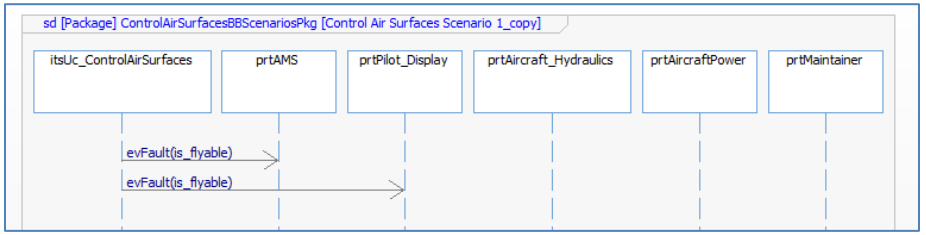
The implementation of the operations **Insert_Flyable_Faults**, **Insert_Unflyable_Faults**, and **removeFaults** for the **Uc_ControlAirSurfaces** block is straightforward:





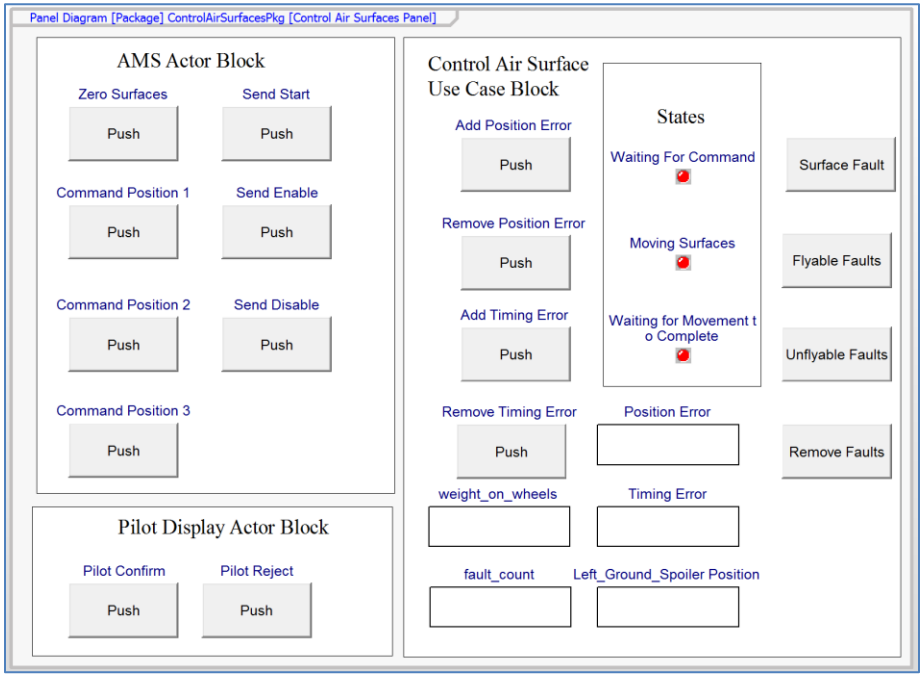
The first operation, **Insert_Flyable_Faults**, adds a fault to the lower rudder and the **Left_Inboard_Aileron** – this still leaves a flyable set. The second operation, **Insert_Unflyable_Faults**, sets faults to both the inboard and outboard aileron on the left side – an unflyable situation. The **Remove_Faults** operation just sets the fault status of these surfaces to **FALSE**. Finally, the **setFault_S**tatus operation of the **CAS_Surface_Positions** block sets the fault in the specified control surface.

As before, don't forget to add the **evFault** event to the interface blocks (and the actor blocks) as a *directedFeature* required in the interface and offered in the actor blocks. Although we've been adding this by manually editing the interface block, there is another way: Create a new sequence diagram with that sends the event evFault from the use case to the actor blocks. The easiest way to do that is to copy one of the existing sequence diagrams in the **ControlAirSurfacesBBScenariosPkg** package, remove all the messages from it and add the events to the appropriate lifelines, thusly:



Now, right click on white space in the diagram and select *SE-Toolkit > Port and Interfaces > Create Ports* and Interfaces. This will add the event to the interfaces.

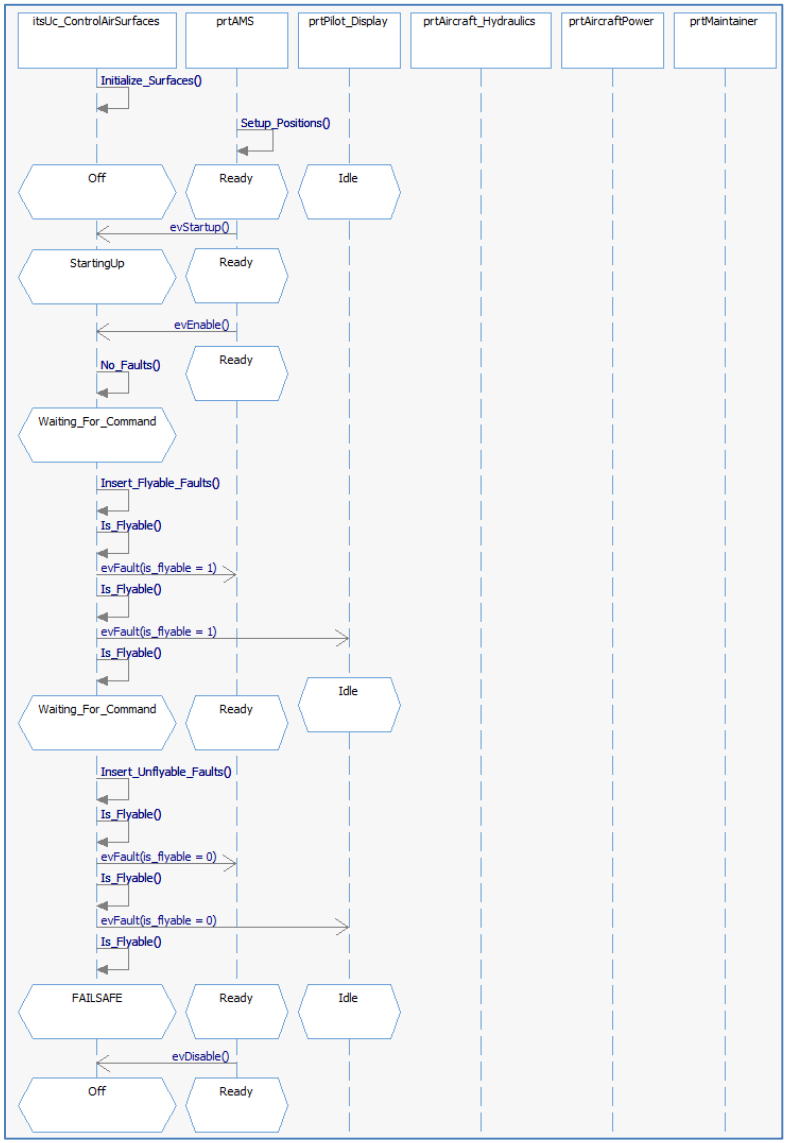
Lastly, we can update the Panel Diagram so that we can generate the **evSurfaceFault** event. As before, we recommend all the events that use the interface should be dragged to the **ControlAirSurfaceInterfacesPkg**.



Let's run a few scenarios for this iteration of this use case analysis model. Run the simulation to get to the **Waiting_For_Command** state, then

1. Set flyable faults and generate the **evSurfaceFault** event
2. Set unflyable faults and generate the **evSurfaceFault** event

Here is the outcome:



At this point, we've completed the functional analysis of the two use cases **Start Up** and **Control Air Surfaces**. The former was analyzed using a flow-based workflow with an activity diagram; then we derived sequence diagrams from that and created ports and interfaces to support simulation.

We also did some safety analysis. Finally, we created and executed the state machine for the use case allowing us to simulate the requirements model to identify and correct requirements defects and omissions.

The latter use case was analyzed with a scenario-based approach. We started with sequence diagrams, then did a data model, safety analysis and a few iterations of state machine creation and execution.

Now we're ready to look towards architecture. We'll start with some architectural analysis of alternatives.

8 Case Study: Architectural Analysis

The purpose of architectural analysis is many-fold but in this section we focus solely on the *analysis of alternatives*; that is, we will analyze different architectural or technology choices to determine the best choice for the needs of the specific system under development. This is also known as *trade study analysis.* The workflow for architectural analysis was shown previous in Figure 5 and Figure 6.

In this section we well apply this process to create an optimized architecture for our system, understanding that the architecture is incomplete because we have only considered two of the use cases in this iteration. Some of these steps will be assisted with the SE Toolkit automation functions. It is important to understand that there are other, even more rigorous ways to support the evaluation of alternatives. These include the use of Rhapsody's parametric constraint evaluator (PCE) profile. We will be applying a slightly simplified method that is practical and, for most purposes, rigorous enough to meet the need.

8.1 Identify Key System Functions

As we pointed out in Section 4.2 on page 16, key system functions are system functions that are important, architectural, and subject to optimization. A system function that is important but neither architectural nor subject to optimization need not be analyzed for trade offs. To be optimizable, in this case, means the selection of a different architectural structure or different technology can result in significant benefit. For example, if you want to provide motive force for a robot arm, should you use pneumatics, hydraulics, or an electrical motor? All have pros and cons, and a trade study can select which is best for the given system given its requirements, contraints, and usage context. However, technology choices that only affect a single engineering domain (such as electronics design) should be deferred and made by the relevant downstream engineering team. It is particularly important to use trade studies when the impact of a technical selection is manifest across multiple engineering disciplines or across multiple subsystem teams.

This can be subtle. For example, requiring functionality be done in a certain way in software may greatly impact the need for available memory and computational resources, affecting the electrical architecture. The communications media among subsystem is another source of multi-disciplinary concerns. Internal communication bus selection is an electronics decision but impacts software performance and throughput and well as cable management, a mechanical concern.

How to find System Functions

System functions show as actions performed by the system on activity and/or state diagrams or as services invoked on sequence diagrams. In the latter case, they are usually manifested as "messages to self" on the use case lifeline.

This use cases we examined require the following kinds of system functions:

- control of surface movement
- measurement of surface movement position
- measurement of surface movement timing
- error date storage
- checking power status
- checking hydraulic status
- checking software integrity
- communicating with the aircraft AMS, Pilot Display, Power, and Hydraulic systems (presumably they have an already defined interface).

In this case, we will focus on the movement of the control surface. Mostly, this is done through the application of hydraulic force provided by the aircraft hydraulic system. The basic schematic is shown in Figure 144. The hydraulic pressure results in a positive movement of the control surface mediated through the movement of a piston and a connecting element. Negative movement is performed by changing the position of the selector switch and applying pressure.

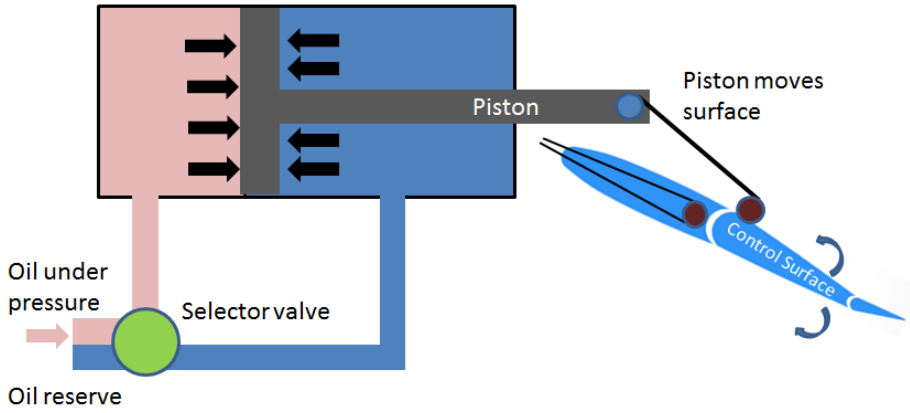


Figure 144: Simplified hydraulic schematic

However several of the control surfaces have *trim tabs.* These are smaller control surfaces that are used to fine tune the aerodynamic effect of the control surface. We've called them out as independent surfaces but they are really subcomponents of the basic control surface.

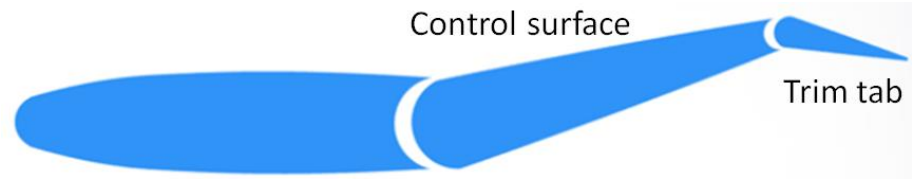


Figure 145: Control Surface with Trim Tab

In addition, some control surfaces extend forward and retract backwards. While the primary motive force (hydraulics) has been determined for the primary control surface, how the trim tab and extension/retraction mechanism works is not yet decided. This will be the focus of our trade study.

8.2 Define Candidate Solutions

In this case we will consider two different methods for moving the trim tabs and extension of the surface:

- Hydraulic force
- Electric motor
- Self-contained electrohydaulic unit for each control surface

The first case will require additional fluid cabling and hydraulic actuators. Schematically, that solution looks something like Figure 146 for trim tab control and Figure 147 for exension and retraction.

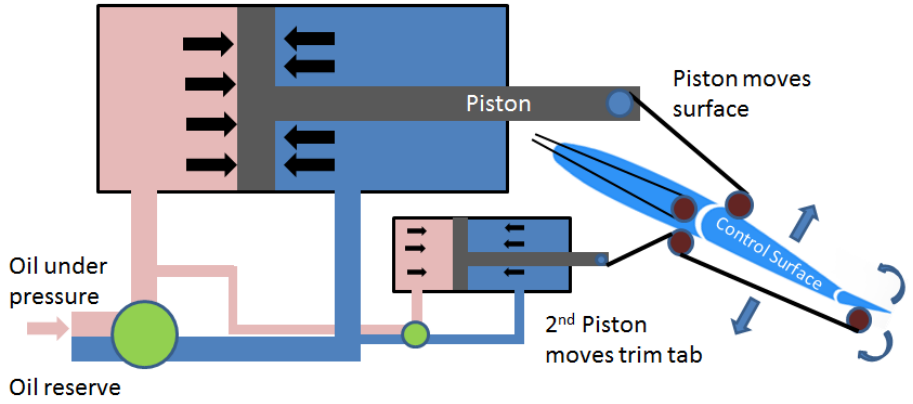


Figure 146: Hydraulic control of trim tab

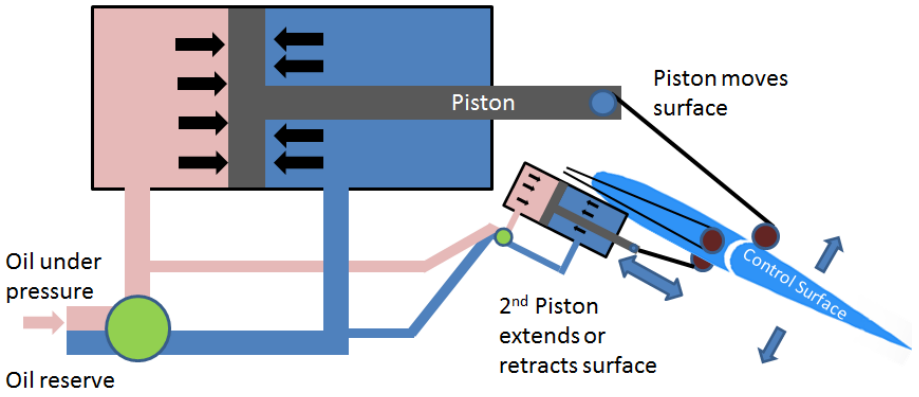


Figure 147: Hydraulic control of extension/retraction

Remember that a control surface, optionally, has either a trim tab or extend and retract, but never both.

The second case will involve installing small electric motors near each control surface and some small additional cabling for electric power (note that power and communications cabling is already required to support measurement of movement).

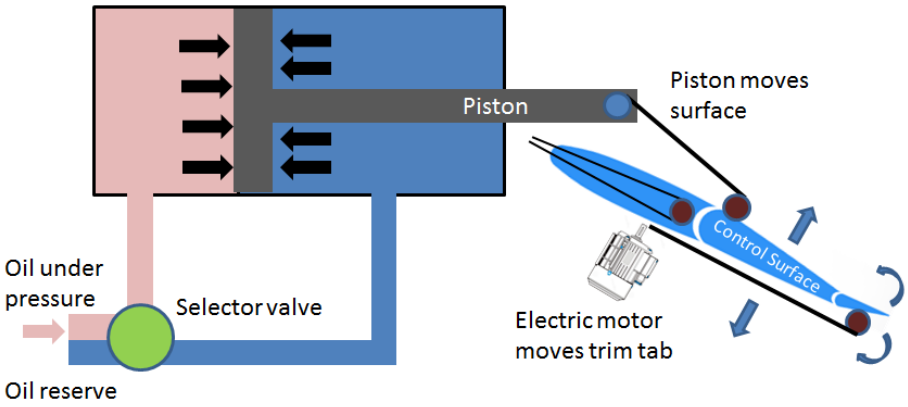


Figure 148: Electric motor control of trim tab

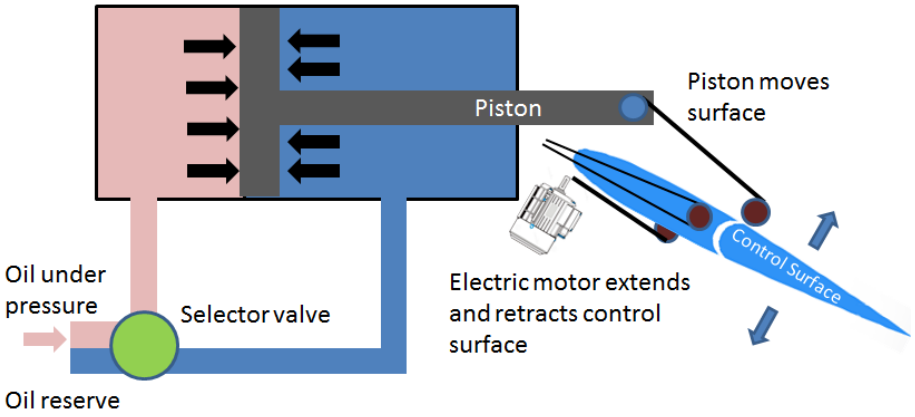


Figure 149: Electric motor control of extension/retraction

The third solution is to use off-the-shelf self-contained electrohydraulic units at each control surface trim tab and extension point.

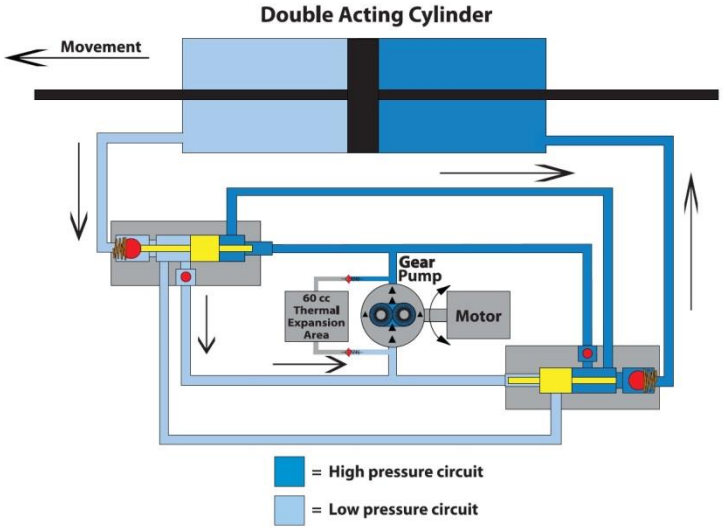


Figure 150: Electrohydraulic Actuator

In use, it is placed much as is the motor in Figure 148 and Figure 149.

We can model these solutions are different subclasses of the generic system functions. To do this:

- In the **DesignSynthesis::ArchitecturalAnalysisPkg** package create a new package; **TrimControlTradeStudy**.
- In the new package, add a new block definition diagram named **Trim Control Alternatives**.
- On this diagram, add new blocks:
  - **PositionControl**
  - **TrimControl**
  - **HydraulicTrimControl**
  - **ElectricTrimControl**
  - **ElectriHydraulicTrimControl**

- **Extensioncontrol**
- **HydraulicExtensionControl**
- **ElectricExtensioncontrol**
- **ElectroHydraulicExtensionControl**

- The **PositionControl** block has two operations that are aspects of this: Add
  - **moveTo(x: int)**
  - **zero()**
  - **ValidateCommand(x: int)**
- Add the generalization relations, as in Figure 151

Generalization means "is a kind of", so this relationship is important as these different technical solutions are specific realizations of the more generic system functions **PositionControl**.

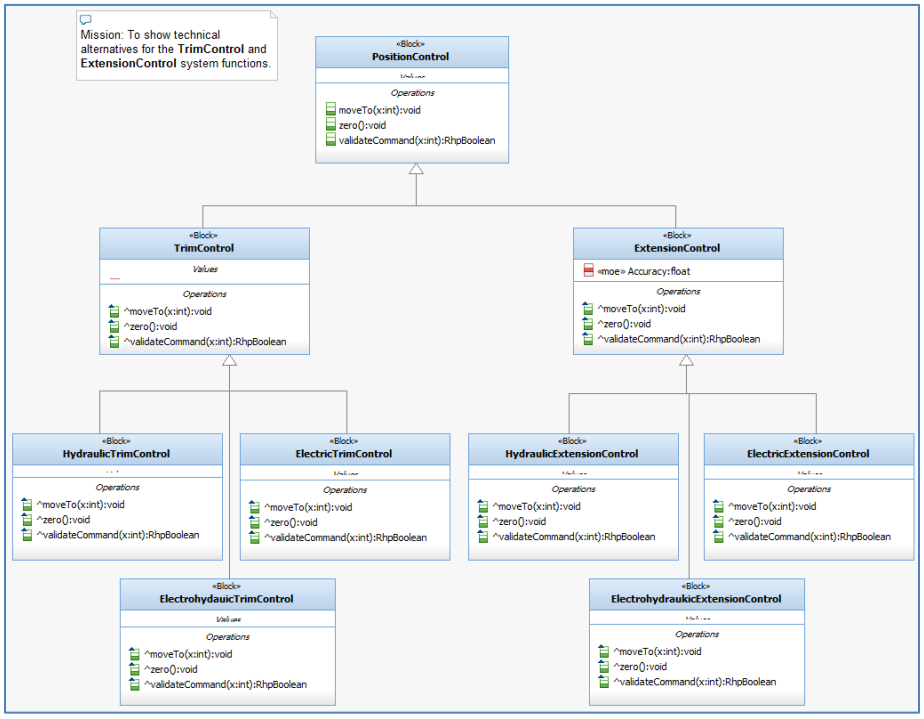


Figure 151: Modeling the candidate solutions

Note that in Figure 151, we used the display options to show inherited operations (indicated with the "^" symbol). This is optional, but we believe that it adds clarity in this circumstance.

It is important to note that these proposed solutions might differ in important qualities of service, including safety, reliability, and security. The proposed solutions and their quantified properties should take these aspects into account (they can even be direct assessment critieria). This means that in real life, the solutions must subjected to dependability analysis as a part of the analysis of alternatives.

8.3 Architectural Trade Study: Define Assessment Criteria

The key to selecting one technical solution over another is the identification of the assessment criteria. Good assessment criteria allow us to distinguish between good and better solutions in how they effect important, measureable properties of the system. In our case, there are five assessment criteria:

- Accuracy
- Weight
- Reliability
- Parts Cost
- Maintenance Cost

Add these to the **PositionControl** block as attributes (of type *float* or *double*), and then *in the browser*, select all attributes and *Change To* an **moe**. **moe** is a new metaclass (in Rhapsody, a "New Term Stereotype") defined in the HarmonySE profile. It brings along a *tag* called **weight**.

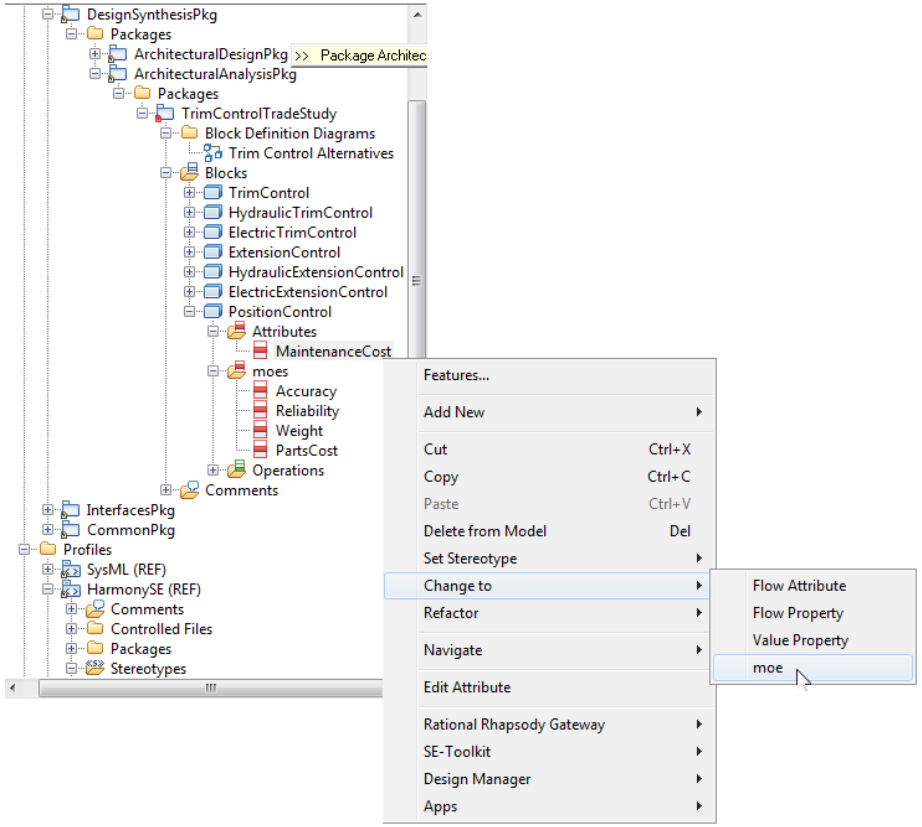


Figure 152: Changing an attribute to an moe

If you've turned the display options of the attributes/value properties on in the diagram for the **PositionControl** block, it should now look like this:

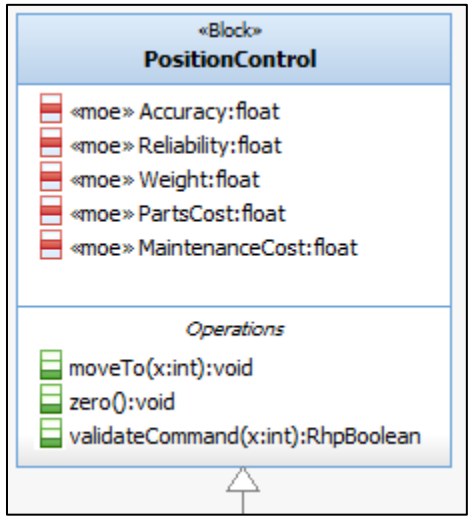


Figure 153: PositionControl with moes added

8.4 Architectural Trade Study: Assign Weights to Criteria

We will assign these MOE values that indicate the degree to which each of the specific solutions optimizes that property. We will scale these so that they are in the range of 1 to 10. We will assign the weights of each MOE to identify its relative importance. The **weights** will be normalized so that they sum up to 1.00.

These MOEs, like any attribute, are inherited in all the specialized subclasses of **PositionControl**. That means that each subclass will have all the MOEs, but will not inherit default values nor values of the **weight** tag. We will assign the default values for each of the subclasses to provide the information as to the degree to which that specific technical solution optimizes that MOE. The weights won't change in the subclass hierarchy; however, since the values of the tags are not inherited, the SE Toolkit provides a tool to copy these values down to the subclasses.

Let's assign the weights first.

The weighting value is an assessment of the relative importance of that specific criterion to the overall "goodness" of the solution. The higher the weight, the more crucial it is. Normalization (so that the sum of all weights equals 1.00) is a common method use do ensure reasonable relative weighting factors. In this case we'll make the following assignments

- Accuracy: 0.30
- Weight: 0.20
- Reliability: 0.25
- Parts Cost: 0.10
- Maintenance Cost: 0.15

To assign these, double click on each MOE in the browser, go to the Tags pane and assign the value:

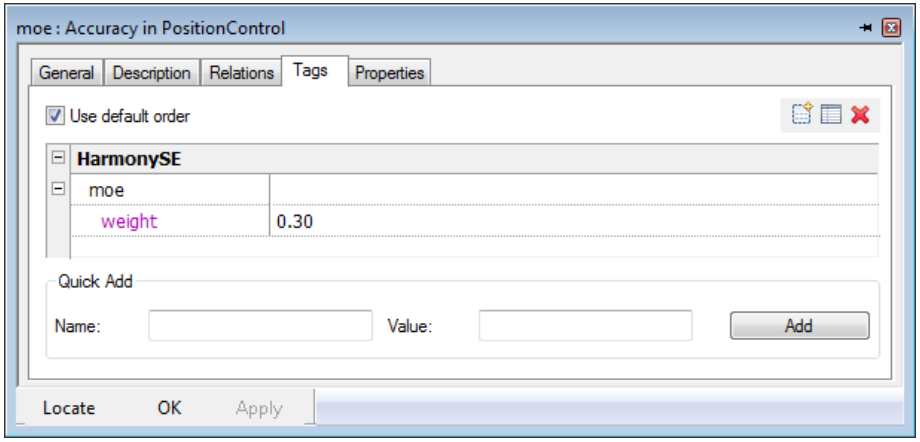


Figure 154 shows the MOE weighting factors on the diagram. To see this, drag the MOEs from the browser, and then right click on each, select *Display Options*, go to the *Compartments* pane and click on the *Customize* button. There you can add the compartment to show the tags.

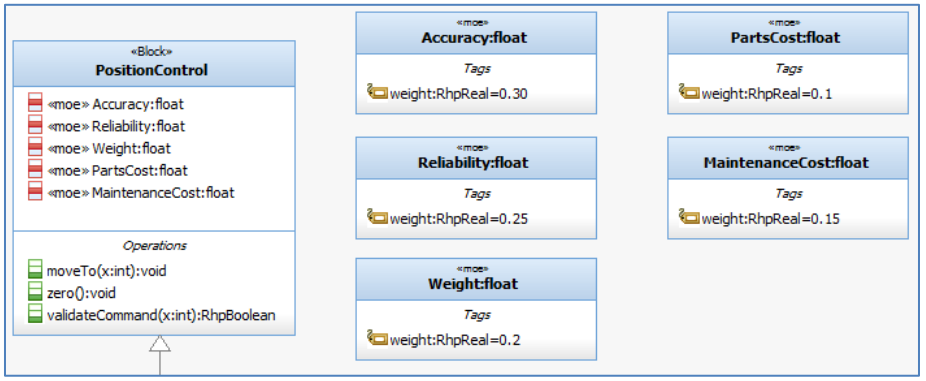


Figure 154: Showing moe weights

To copy these down to the children, right click the **PositionControl** block and select *SE-Toolkit > TradeStudies > Copy MOEs to Children*. In this, slightly unusual case, you'll have to repeat the procedure for the **TrimControl** and **ExtensionControl** blocks, as this helper only works with the immediate children of a block. If you now inspect those subclasses, such as **ElectricTimControl**, you will see that it also has the set of MOEs with the correct values assigned to the weights.

8.5 Architectural Trade Study: Define Utility Curve for Each Criterion

The utility curve computes a "goodness" score based on a quantitative value associated with the solution. The utility curve can be any shape but, by far, those most common is the "linear utility curve." This curve is a straight line defined by two points. The first point for this MOE is the worst candidate solution being considered has a utility value of 0 while the best candidate being considered has a value to 10. Given these two points, (worst, 0) and (best, 10), a line can be constructed going through both. This is the linear utility curve.

The equation for a line, given two points (x1, y1) and (x2, y2) is simply

$$y = \frac{y2 - y1}{(x2 - x1)}x + b$$

We have special conditions, such (worst, 0) and (best, 10) on the line. This simplifies the utility curve to

$$moe = \frac{10}{best - worst}CandidateValue + b$$

And

$$b = -\frac{10}{best - worst}worst$$

Where

- *best* is the value of the criterion for the best candidate solution
- *worst* is the value of the criterion for the worst candidate solution

For example, let's consider a system where our criterion is *throughput*, measured in messages per second. The worst candidate under consideration has a throughput of 17,000 messages/second and the best candidate has a throughput of 100,000 messages/second. Applying our last two equations provides a solution of

$$moe = \frac{Throughput}{8300} - 170/83$$

A third candidate solution, that has a throughput of 70,000 message per second would then have a computed MOE score of 6.3855.

Note: There are lots of other ways to construct utility curves for trade study analysis. Interested parties are encouraged to look up references for specific methods.

The next step is to construct the equations for each MOE using this approach. For the purpose of this example, assume the following sets of values are true for the set of criteria. In actual practice, this data would come from lab measurement, manufactured specs, historical data, or estimation.

Table 1: Trade Study Criterion Values

Solution/moe	Accuracy (mm)	Weight (kg)	Reliability (mtbf hrs)	Parts cost ($)	Main. Cost ($)
Hydraulic	5	72	4000	800	2000
Electric	1	24	3200	550	2700
Electrohydraulic	2	69	3500	760	2100

Using the method outlined above results in the following set of equations:

$$accuracyMOE = -\frac{5}{2}accuracy + \frac{25}{2}$$

$$weightMOE = -\frac{5}{24}weight + 15$$

$$reliabilityMOE = \frac{reliability}{80} - 40$$

$$partCostMOE = -\frac{partsCost}{25} + 32$$

$$maintenanceCostMOE = -\frac{maintenanceCost}{70} + \frac{270}{7}$$

8.6 Architectural Trade Study: Assign MOEs to Candidate Solutions

The equations for MOEs can be captured in SysML parametric diagrams.

- In the browser, right click on the **TrimControlTradeStudy** package and select *Add New > Diagrams > Parametric Diagram*. Name this diagram, **Trim Control Trade Study Parametrics**.

- Drag the **PositionControl** block onto the diagram, then drag each of its MOEs to inside the **PositionControl** block on the diagram.
- Add a *ConstraintProperty* from the toolbar onto the diagram. Name this *ConstraintProperty* **TrimControlMOEs**. Size this box to be the same height at the **PositionControl** block.
- Add *ConstraintParameters* to the left edge of the constraint property:
  - **accuracy**
  - **weight**
  - **reliability**
  - **partsCost**
  - **maintenanceCost**
- Add a *BindingConnector* between each constraint parameter and the corresponding attribute in the **PositionControl** block.
- Using the technique outlined above, add the equation for each computed MOE, as constraints in the **TrimControlMOEs** constraint property.
  - **accuracyMOE**
  - **weightMOE**
  - **reliabilityMOE**
  - **partCostMOE**
  - **maintenanceCostMOE**
- Now add a *ConstraintParameter* for each of these computed MOEs with the same name as in the previous step
- Add a new *ConstraintProperty* named **TrimControlObjectiveFunction** and add constraint parameters that match the ones in the previous step
- Connect the matching constraint parameters between the two *ConstraintProperties* with binding connectors
- Add the objective function as a constraint, computing the objective function as the weighted sum of the property times its weighting factor (stored in the **weight** tag)

Note, you can make the constraints visible by right clicking on the *ConstraintProperty* and selecting *Display Options*. Then go to the *Components* pane and click *Customize*, and add *Constraints* to the list.

Once you're done, you shound have a diagram that looks like Figure 155.

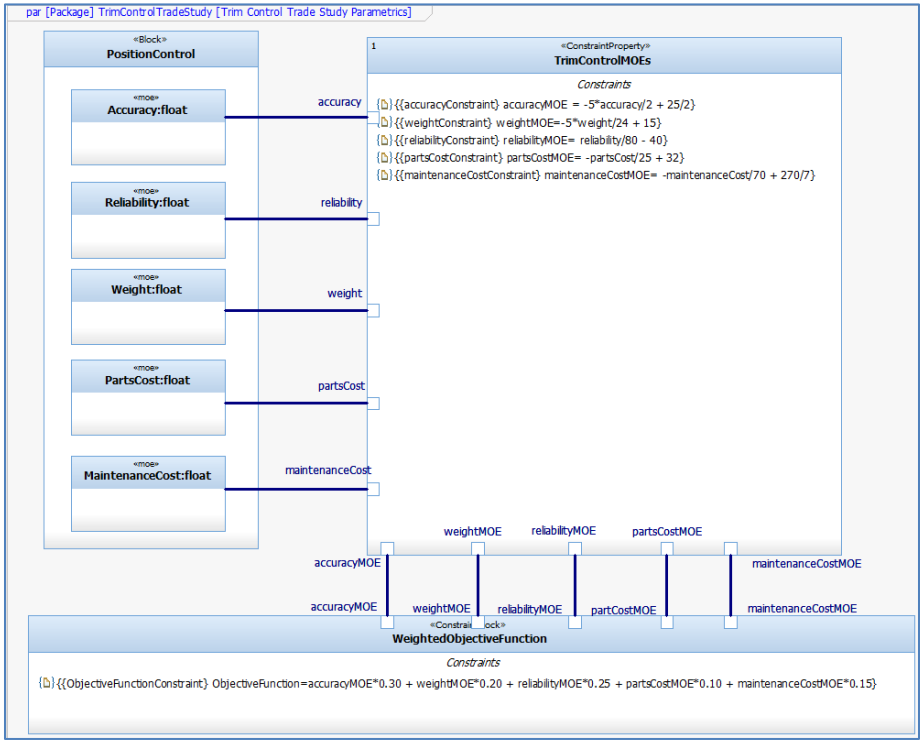


Figure 155: Parametric diagram for trade study analysis

Rhapsody provides a *Parametric Constraint Evaluator (PCE)* profile that connects to third-party mathematical computational engines to perform the calculations for the three solutions. However, we will do a slightly simpler approach using the facilities of the SE Toolkit. It will use Microsoft Excel as the computational engine for evaluation of the constraints.

Build a Solution Architecture Diagram

First, let's build a *Solution Architecture Diagram*. This is a block definition diagram that shows the alternative solutions.

- In the **TrimControlTradeStudy** package, add a new Block Definition Diagram. Name this diagram **Trim Control Solution Architecture**.

- Add blocks representing the alternative solution architectures
  - Block **HydraulicTrimControlSolution**
  - Block **ElectricTrimControlSolution**
  - Block **ElectroHydraulicTimeControlSolution**
- Drag the six solution blocks onto the diagram from the browser
  - **HydraulicTrimControl**
  - **HydraulicExtensionControl**
  - **ElectricTrimControl**
  - **ElectricExtensionControl**
- Make the appropriate composition relations among the blocks
  - **HydraulicTrimControlSolution** is composed of **HydraulicTrimControl** and **HydraulicExtensionControl**
  - **ElectricTrimControlSolution** is composed of **ElectricTrimControl** and **ElectricExtensionControl**
  - **ElectroHydraulicTrimControlSolution** is composed of **ElectroHydraulicTrimControl** and **ElectroHydraulicExtensionControl**
- Compute the MOE value by applying the MOE equations to the values from Table 1 of the appropriate solutions
  - Assigning the values for the best and worst scores is easy: it's either 0 or 10, because that's how we defined the linear utility function. To determine the scores are between the best and worst, you'll have to solve the equations above.
  - For example, to determine the value of the *MOE* **Accuracy** of the **Electrohydraulic Trim Control** solution, take the value of the accuracy of the solution from Table 1 (2), compute the *MOE* by using the **accuracyMOE** equation, and assign the result (7.5) to the value of the **Accuracy** *MOE* in the **ElectrohydraulicTrimControl** and **ElectroHydraulicExtensionControl** blocks.

Your diagram should look something like Figure 156.

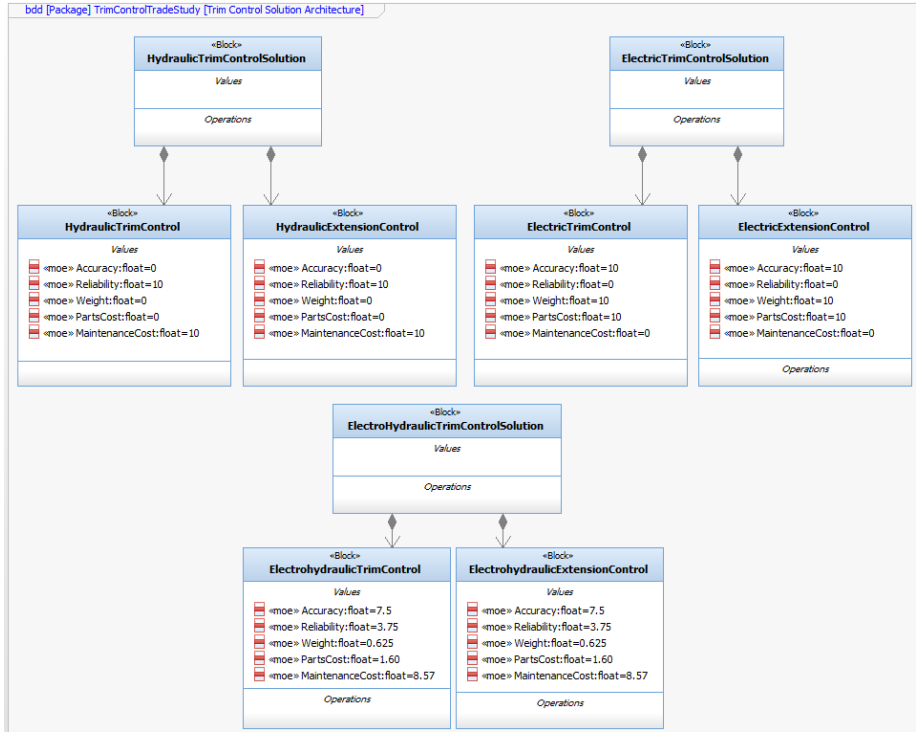


Figure 156: Trim Control Solution Architecture

8.7 Architectural Trade Study: Determine Solution

Construct an Option Analysis Diagram

Next, make another block definition diagram in the same package named **Trim Control Option Analysis**. Drag the three potential solution architecture blocks on to it: **HydraulicTrimControlSolution, ElectricTrimControlSolution** and **ElectrohydraulicTrimControlSolution.**

This diagram is very simple and provides a context for the SE-Toolkit to do the analysis:

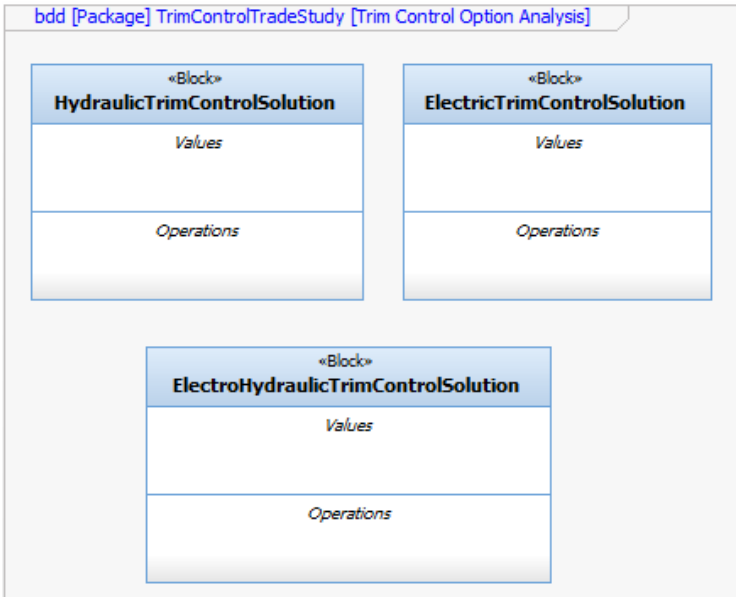


Figure 157: Trim Control Option Analysis Diagram

Right click in this diagram and select *SE-Toolkit > Trade Studies > Perform Trade Analysis*. The toolkit will create a new *Controlled File* named **Trim Control Option Analysis_TradeStudy.xls**. Double-clicking this file will open it in Excel and show you the trade analysis with the computation of the objective function performed by Excel:

		HydraulicTrim ControlSolution		ElectricTrim ControlSolution		ElectroHydraulicTrim ControlSolution	
	weight	value	WV	value	WV	value	WV
PositionControl.Accuracy	0.3	0	0	10	3	7.5	2.25
PositionControl.Reliability	0.25	10	2.5	0	0	3.75	0.9375
PositionControl.Weight	0.2	0	0	10	2	0.625	0.125
PositionControl.PartsCost	0.1	0	0	10	1	1.6	0.16
PositionControl.MaintenanceCost	0.15	10	1.5	0	0	8.57	1.2855
			4		6		4.758

Figure 158: Computation of the objective function

By this analysis, the electric motor solution is our best choice, since it has an objective function value of 6., versus 4 for the purely hydraulic solution and 4.758 for the self-contained electrohydraulic units.

8.8 Merge Solutions into System Architecture

Because this is the first iteration, we don't have an existing subsystem architecture into which to insert the results of our trade study. When we get to architectural design (next), we will insert the solution where it makes sense. In some cases, the solution at this point is obvious as we've identified a subsystem. However, in this case, we've identified a subcomponent of one or more subsystems, so we will defer the merging the solution into the architecture until we've identified where it should go.

9 Case Study: Architectural Design

In architectural design, we will

- Identify the subsystems
- Allocate requirements and use cases to subsystems
- Define the interfaces and flows between the subsystems
- Derive subsystem requirements
- Update the logical data schema
- Update the dependability analysis
- Create the system verification plan

The workflow for the Architectural Design activity is shown in Figure 10 back on page 19. We won't explore some of these tasks to save space, including *Develop Control Laws* and *Analyze Dependability*. We will do at least some of the work associated with all the other activities and tasks from Figure 10.

The first thing we'll do is to merge in the features from the various use case blocks

9.1 Identify Subsystems

A **subsystem** is a large-scale architectural element that

- Meets a common set of requirements (**coherence**)
- Contains elements that interact strongly (**tight coupling**)
- Contains elements that interact weakly with other subsystem (**independence**)
- Hide internal structure and implementation detail (**encapsulation**)
- Provides or requires well defined sets of services (**interfaces**)
- Typically, developed by a single team (**common developers**)
- Usually contains aspects from multiple engineering disciplines (**interdisciplinary**)

Good subsystems are

- Coherent (together provide a small number of purposes)
- Internally tightly coupled
- Externally loosely coupled (with other subsystems and their components)
- Collaborative in the architecture with via a small number of well defined interfaces

In thinking about this system, it is clear that we need several different kinds of structures to provide sets of coherent services. We'll start by creating a block definition diagram showing our basic idea for the architecture.

- In the *DesignSynthesisPkg > ArchitecturalDesignPkg*, add a new block definition diagram. Name this diagram **ACES System Structure**.
- Add a system block named **ACES** (if one does not already exist)
- Add the following subsystems as blocks
  - **ACES_Management**
  - **ACES_Hydraulics**
  - **ACES_Power**
  - **ACES_ControlSurface**
  - **ACES_ControlSurfaceWithTrim**
  - **ACES_ControlSurfaceRetracting**

Connect the first four to the **ACES** system block with composition relations and make the last two blocks subclasses of the **ACES_Control_Surface** block. See Figure 159.

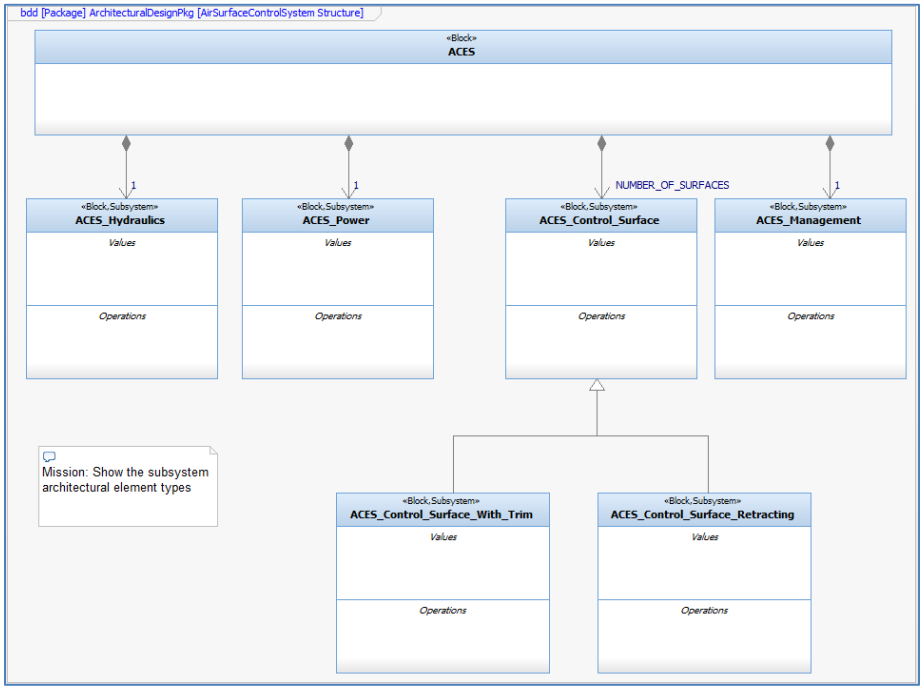


Figure 159: Subsystem composition architecture

This architecture takes advantage of the similarities between the three kinds of control surfaces – simple control surfaces, control surfaces that also have trim tabs, and control surfaces that retract and extend.

We now need to create packages for each of the subsystems. Fortunately, there's an SE Toolkit feature for that. Right click the **ACES** block on the diagram and select *SE-Toolkit > Architecture Tools > Create Sub Packages*. This wizard will mark the subsystems with the stereotype «Subsystem», moves the block to its package, and adds a tag **isSubsystem** with the value *TRUE* (used later in the hand off workflow).

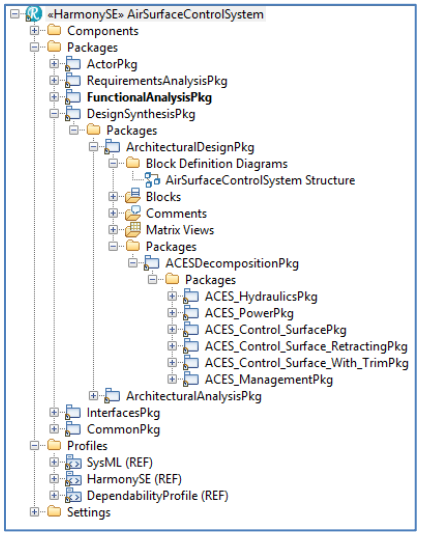
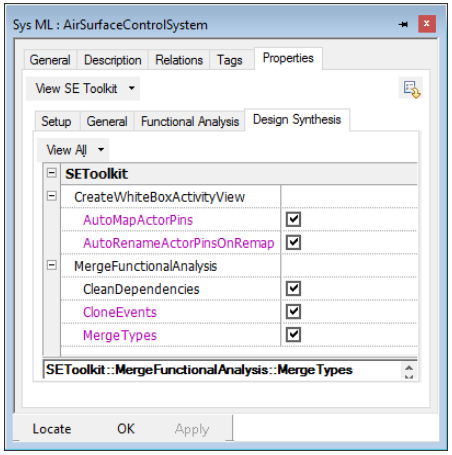


Figure 160: Added subsystem packages

9.1.1 Merge functional analysis

Note: before you apply the SE-Toolkit feature you'll want to set the properties for the SE Toolkit to *Clone Events* and *Merge Types*. To do this, select the project in the browser, double click to open the *Features* dialog, go to the *Properties* tab, *View* the SE-Toolkit properties and click the checkboxes as shown.



In this step, the features of the use case blocks are merged into the system block so they can be allocated into the appropriate subsystems. The SE Toolkit provides a tool to do that. Right click on the **ACES** block in the browser or on the diagram and select *SE-Toolkit > Architectural Tools > Merge Functional Analysis*. This tool will collect up the attributes and services from the various use case blocks in the functional analysis package and add them to the **ACES** block.

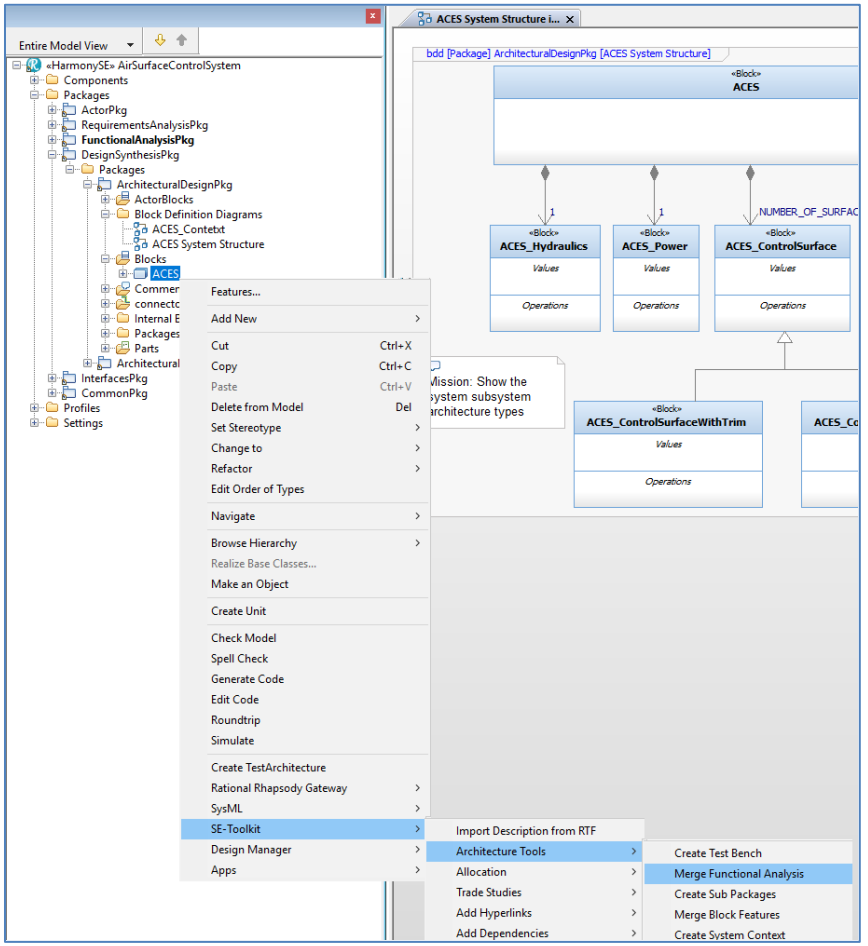


Figure 161: Merge Functional Analysis tool

The tool may report finding errors; these are usually because the tool already added in a feature of the same name from some other use case block. This is an indication that you should look at the merged feature to make sure that it properly merges the features from all relevant use case blocks.

Issues with Merging Functional Analysis

When you merge from multiple use cases there are several cases that must be considered:

1. The semantic feature is unique to one use case
2. The semantic feature occurs in exactly the same form in multiple use case
3. The semantic feature has different names but is meant to be the same feature
4. The feature has the same name in different use cases but intended to be semantically distinct
5. The feature occurs in multiple use cases but is different in form
    a. Same name, different properties
    b. Different name, different properties but nevertheless still describes the same feature

The term *properties,* in this context means things like the argument list order, type and naming for operations and event receptions, service type (operation, event reception, triggered operation reception), or type (if an attribute or value property), and the value of any stereotype tags that might indicate subrange or qualities of service.

Cases 1 and 2 are the easiest. Simply add the feature to the system block. However, care must be given to ensure that when you think you have case 2 you do not actually have a case 4.

The other cases are more difficult and require human intervention.

Case 3. This often occurs because different use case developers are likely specify an event, service, or datum using a different name while referring to the same system feature. One might imagine on use case developer using an event name **Move_To(x,surfacename)** while another use case that also requires movement to use **goto(surface, position)** or even **commandAllPositions(p: PositionSet).** Semantically, the intent of all of these is the same even though the names and parameter lists are different. Human intervention is required to identify this and merge them into a single service in the system block.

Case 4. This occurs less often, but even if you have a naming guideline to use names expressive of intent, it does occur frequently enough. An event such as **configure** might refer to the setting of minimum and maximum positions of a specific control surface or uploading a new software image. Such errors are harder to identify and require a thorough review of the application in the different use cases.

Case 5a. The use of the same semantic service might require different parameters depending on its actual use. For example **evError**, in one context might have to return the location of the error (for system diagnostics and repair), or the severity of the error (for operational decision making), or the date and time of occurrence (for maintenance purposes). One solution is to merge all these needs together into a single service, knowing that in some contexts not all information may be relevant. Another solution is to create different services that carry the data they need based on the context of their use.

Case 5b. This is a variant of Case 5a and is even more difficult to detect, since the name and properties of the service are different. To detect this requires a solid understanding of the relevant source use case analyses.

Beyond these general issues, there are some issues in older versions of the toolkit. The toolkit clones types and events – assuming you set the checkbox in the properties dialog for the SE Toolkit as mentioned before – but older versions may not always resolve references to the cloned elements. For example, the use case model refers to an event

evUpdatePositions(CAS_Surface_Positions* sp)

The current version of the toolkit properly clones the event and updates the event reception but may not update the type of the parameter **sp**. It should refer to the cloned type **CAS_Surface_Positions** in the **InterfacesPkg > MergedInterfacesPkg > UcControlAirSurfacesDataTypesPkg** but instead refers back to the original copy of the block in the **FunctionalAnalysisPkg > ControlAirSurfacesPkg > ControlAirSurfacesTypesPkg**.

This limitation also applies not only to the parameters of event arguments, but also to the types of value properties of cloned blocks and parts of cloned blocks. For example, cloned block **CAS_Status** has a property called **status** of type **CAS_SystemOperationalState**. Although the latter type was

cloned, the **status** value property may not be updated to refer to the cloned copy.

Bottom line: not all references to types (including blocks) may be properly updated to refer to the cloned version. You'll have to manually review each one to ensure that it properly refers to a type in the **InterfacesPkg** and not in the **FunctionalAnalysisPkg** and update where necessary. The toolkit will get you started but there may still be some work to be done.

What to do about it

The upshot of this is to understand that *the merge of information from different use cases to the system block can never, in principle, be a completely automated process*. The SE Toolkit gets you started, but you must still examine and analyze the result to ensure the intent from each use case is preserved in the system block.

> We recommend ongoing reviews between use case teams, to identify and resolve such issues. These "alignment reviews" take place periodically during the parallel development of the multple use cases (and therefore precedes the architectural merge). This will resolve the simpler issues of conflict between the use case teams. Issues like differences in parameter lists of system functions and data structures are harder because these differences are "out of scope" of the use cases.

The best way to do this is a review of each use case feature set, state machine, and interfaces and how each was merged into the system block. Ideally the system architect and a member from each use case team is present in the review of the merged features set. It is best to complete this review and update before moving on to the allocation of the features to the subsystems.

Completing the Merge of Functional Analysis into the Architecture

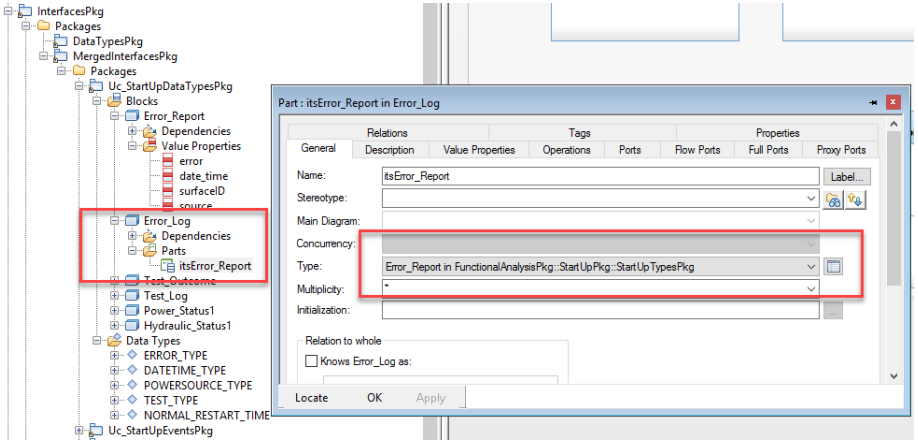
The SE-Toolkit Merge Functional Analysis tool gets the process started. The tool does the following things for you automatically:

- Copies the attributes/value properties and operations (of all types) from each of the use case blocks in the **FunctionalAnalysisPkg** to the identified system block
- Copies all types from the functional analysis use case nested types packages into the **InterfacesPkg > DataTypePkg** into subpackages organized by use case
- Updates the parameter lists of the copies system block functions to refer to the copied types

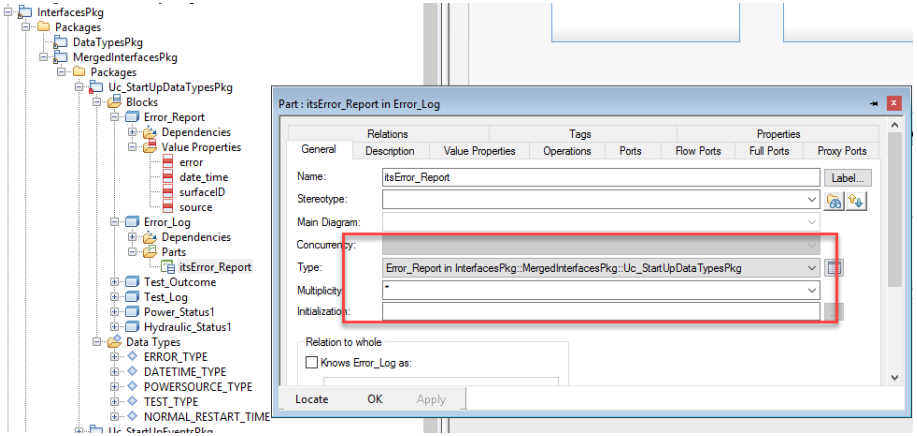
We must now manually complete this merge activity. This is a matter of walking through all the copied elements, updating the names (since they were all name-mangled with the use case name), and merging their semantics, as appropriate. For example, a **evMovementCommand(p: CAS_PositionSet)** and **evMove(surfaceID, position)**, perhaps this becomes a single **evMovement(p: PositionSet)** operation, where we've merged the functionality, and changed the names to remove the use case-specific adornments.

Older Toolkit Version

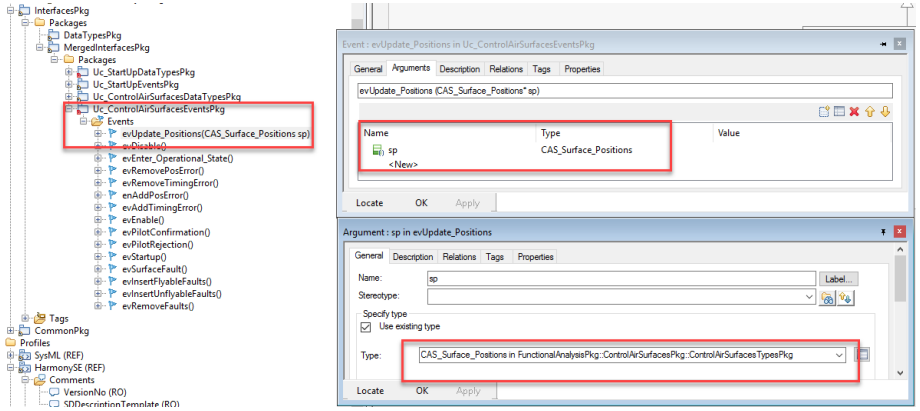
In addition, you should look at the location of each referenced type to be sure that it refers to a type in the **InterfacesPkg** and not one in the **FunctionalAnalysisPkg**. For example, if you see



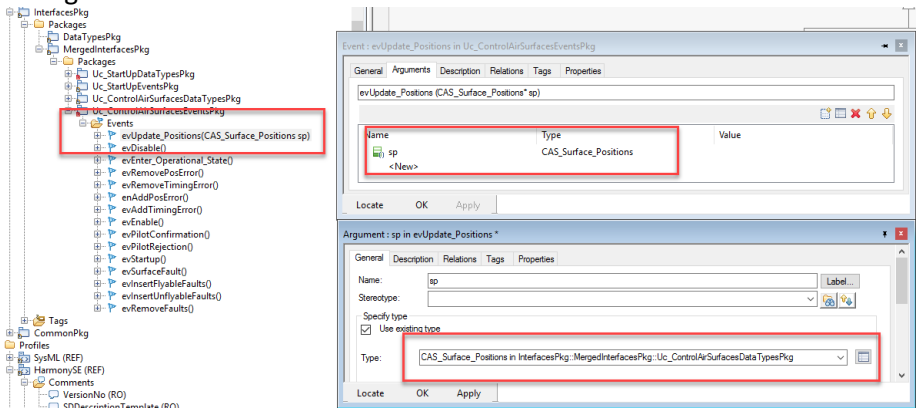
You should change it to



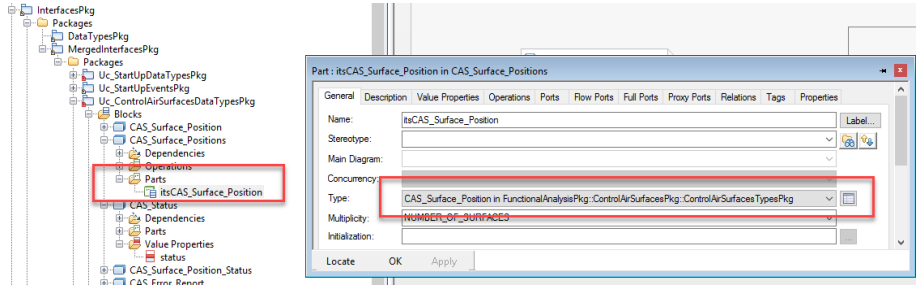
Similarly, for this:



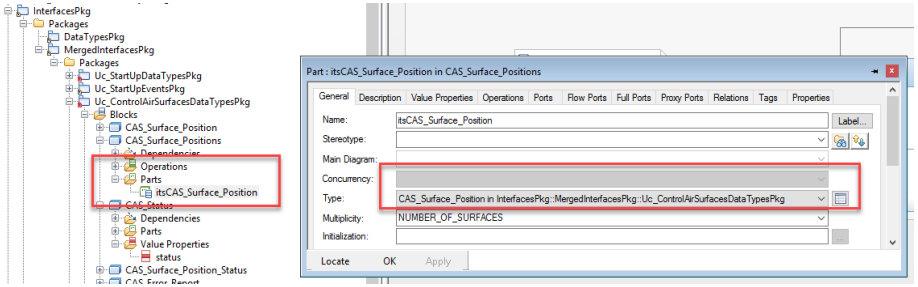
Change it to this:



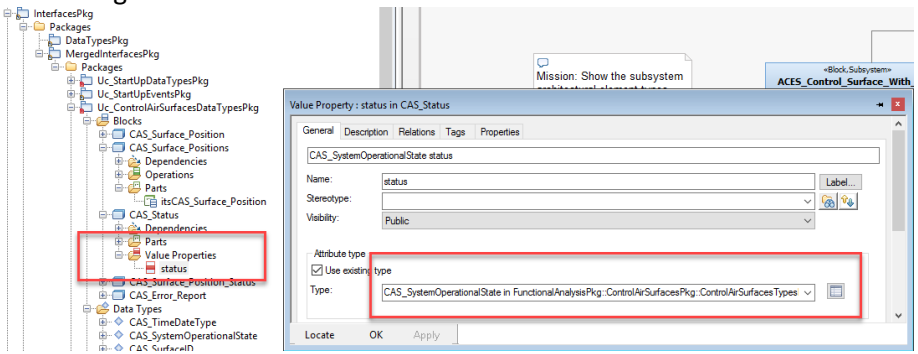
Change this:



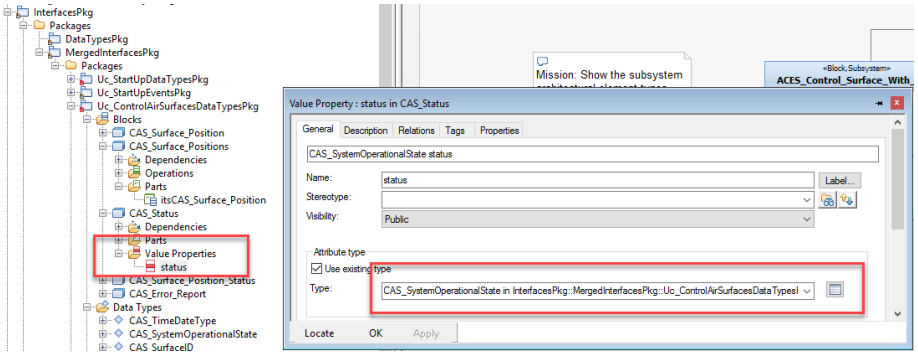
to this:



And change this:



to this:



Repeat for all references to types in the cloned operations, event receptions, blocks, and types in the **InterfacesPkg** and nested packages.

Merging Similar Features

There are other things to merge as well. For example, the two use cases refer to the type of **Surface_ID** differently. The **Start Up** use case used an *int* while the **Control Air Surfaces** use case used an enumerated type named **CAS_SurfaceID**. So any reference to a surface id in an operation, reception, block or type in the **InterfacesPkg > MergedInterfacesPkg > Uc_StartUpxxx** packages should be changed from an *int* to the **CAS_SurfaceID**.

It also makes sense to remove the **CAS_** prefix used for the use case features since now we're in the merged architecture. For example, **CAS_SurfaceID** should become **SurfaceID**. If you change the type name Rhapsody will update all the references that use it for you.

The two use cases also define a date-time type (**DATETIME_TYPE** and **CAS_TimeDateType**), two error types (**ERROR_TYPE** and **CAS_ERROR_TYPE**) and the restart time intervals (**NORMAL_RESTART_TIME** and **NORMAL_RESTART_INTERVAL**). Each should be resolved to a single type used by all relevant cloned elements, and the unused one should be deleted from the **InterfacesPkg**. Since the tookit adds a dependency, drag the dependency from the type to-be-deleted to the type-to-be-retained. In this case, I used the CAS_ versions of all the types. That is, I copied all the dependencies; I changed all the references to the **DATETIME_TYPE** to the **CAS_TimeDateType**; I copied all the enumeration literals from the **ERROR_TYPE** to the **CAS_ERROR_TYPE**; I removed **NORMAL_RESTART_TIME** but kept **NORMAL_RESTART_INTERVAL.** I then went through the blocks and types in the **InterfacesPkg** nested packages and removed all the **CAS_** prefixes.

I also made a pass to identify any merged features of the **ACES** block and events in the **DesignSynthesisPkg** that were there to support simulation, such as the insertion or removal of error conditions. To all these, I added the stereotype **«nonNormative».** If desired, you can remove any merged features stereotyped **«nonNormative»**, since they were just used to

facilitate simulation. If you think they will continue to be helpful, however, feel free to keep them.

With the two use cases defined, the merge results in the following attributes and operation copied from the use cases to the **ACES** block:

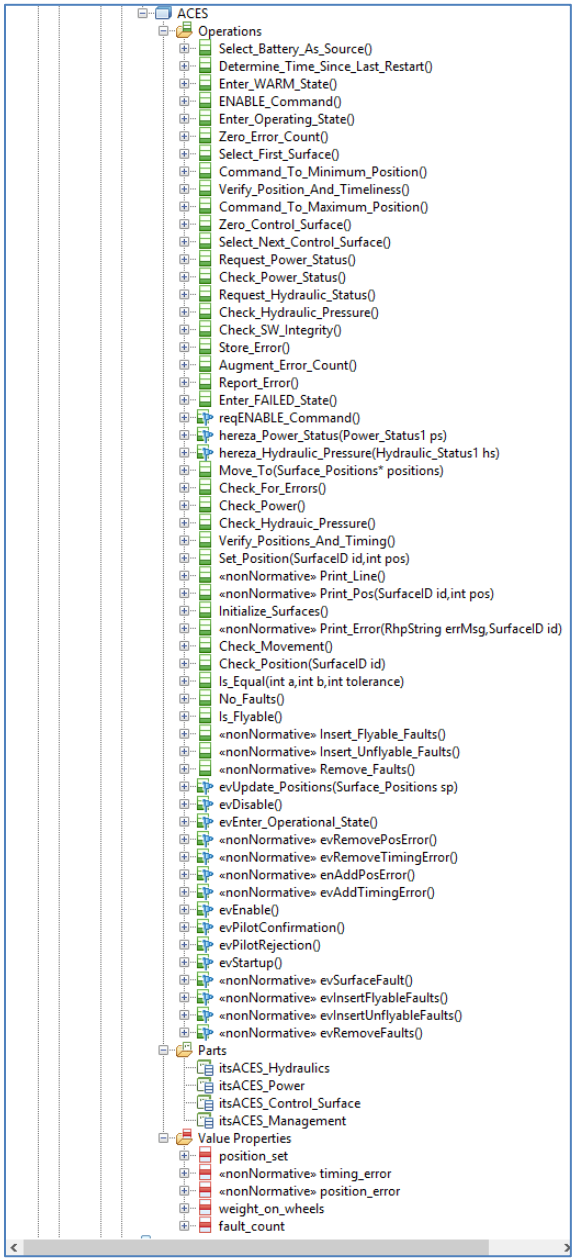


Figure 162: Result of Merge Functional Analysis

I then moved all the blocks and types from the packages nested within the **InterfacesPkg** to the **InterfacesPkg > DataTypesPkg** package. This results in the following structure and set of types and events:

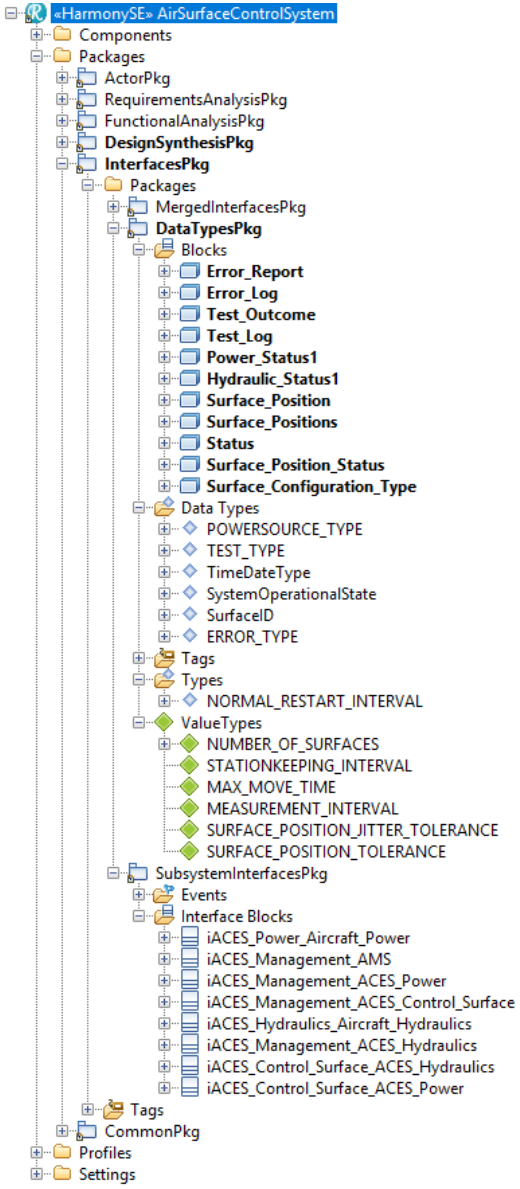


Figure 163: Merged Types and Events

9.1.2 Allocate merged features to subsystem architecture

So all these attributes/value properties and operations/event receptions identified in the functional analysis are merged into the **ACES** system block. What should you do with them next?

These features must be allocated to the subsystems. Many of these features can be directly allocated to a single subsystem but others must be decomposed into subparts which are then allocated. The SE toolkit Allocation Wizard can help out with this task.

Right click on the **ACES** block and select *SE-Toolkit > Allocation > Allocation Wizard* (Figure 164).

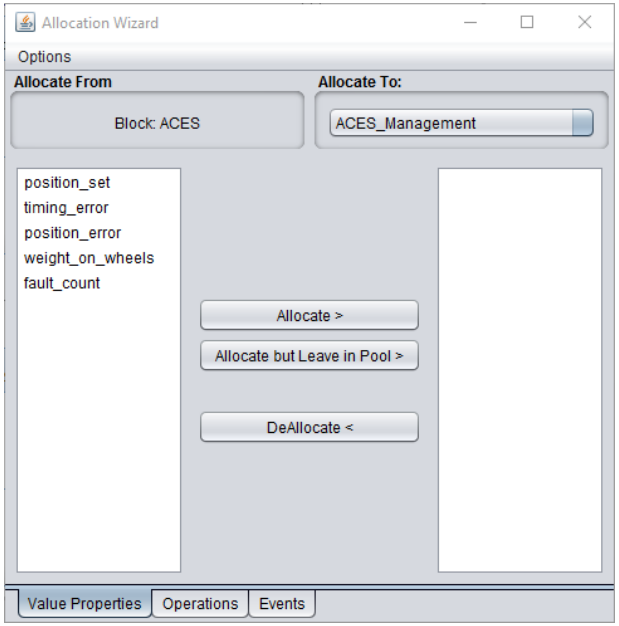


Figure 164: Allocation Wizard

This wizard allows you to allocate attributes, operations and events to different subsystems. The different block features are available as tabs at the bottom of the dialog. The various subsystems are available in a drop down list at the top. By the time you're done, all attributes and event receptions should be allocated and *most* operations. Some operations may result in a set of operations scattered across multiple subsystems and so it may be inappropriate to directly allocate them. For such operations, the Harmony SE profile provides the stereotype **«DecomposedOperation»**. For such operations, add the stereotype by right clicking the operation in the browser and selecting *Set Stereotype > DecomposedOperation*. The allocation wizard will ignore these – meaning that you will have to do the decomposition yourself. Note that some elements may be allocated to more than one subsystem.

The next few figures show the allocations to the **ACES_Management** subsystem. Naturally, elements are allocated to the other subsytems as well. Note that in Figure 166 that some operations remain unallocated. In this case, these are «nonNormative» operations that are there only to support simulation and execution of the functional use case model. Later versions of the SE Toolkit may opt to not even put such elements so in the allocation list.

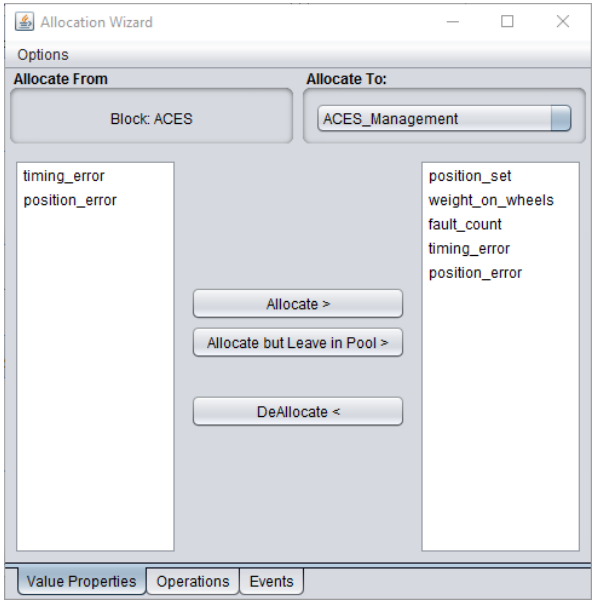


Figure 165: Allocation of attributes to ACES_Management subsystem

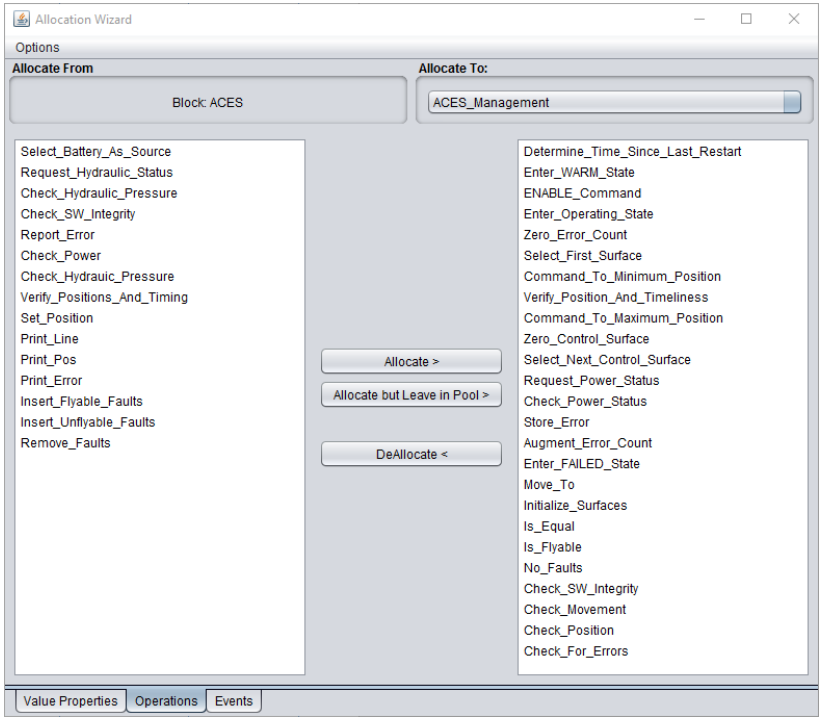


Figure 166: Allocation of operations to ACES_Management subsystem

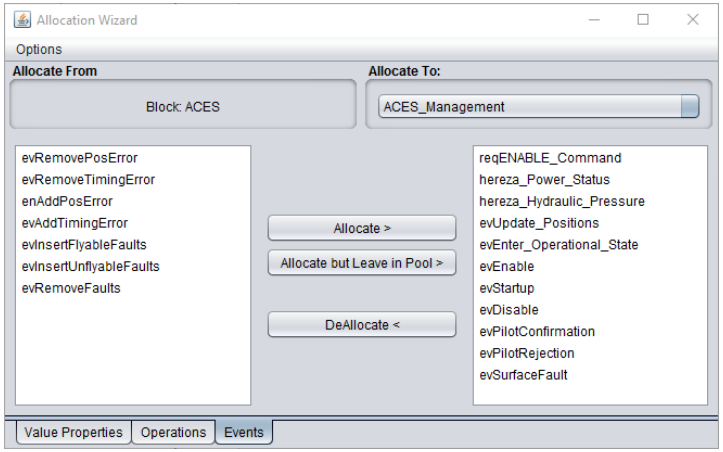


Figure 167: Allocation of events to ACES_Management subsystem

Remember that these aren't all the features that will be allocated to the subsystems; these are only the ones carried over from the use case analysis. As we detail the allocations, we will add additional features to the subsystems by creating white box sequence diagrams.

9.2 Allocate Requirements to Subsystems

Use cases each represent a coherent but limited set of requirements, whereas the system must, in principle, represent all such requirements (at least the ones represented in the current iteration). Requirements must be allocated down into the subsystems that implement them. Some requirements may indeed, be directly allocated to a specific subsystem. Many requirements really specify collective subsystem behavior and so must be decomposed into derived requirements that can be allocated to a given subsystem. Requirements diagrams are a good place to show both the decomposition into derived requirements and their allocation.

9.2.1 Creating Derived Requirements

Since we will be creating derived requirements for the purpose of allocating to subsystems, let's provide a place to put them. Create a **SubsystemReqsPkg** package nested inside **RequirementsAnalysisPkg > RequirementsPkg** (see Figure 168). This package will hold the requirements diagrams for the derivation of the subsystem requirements as well as those requirements themselves.

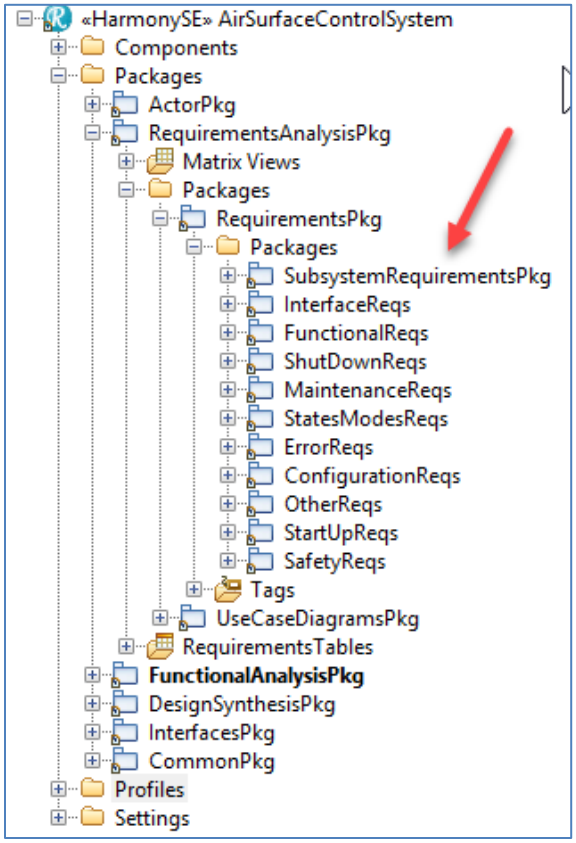


Figure 168: Package for Subsystem Requirements

We'll take this in a couple of phases. First, let's determine which requirements cannot be allocated and must be decomposed into derived requirements. We can represent those derived requirements on diagrams or in a table we construct for that purpose.

Derivation or Derive Requirement?
Both *Derivation* and *Derive Requirements* appear on the Rhapsody Requirements Diagram. The first is provided by Rhapsody as pre-defined stereotype and the latter is defined as a part of the SysML standard. Which should you use? The short answer is "It really doesn't matter but you should be consistent in your model." Both are *New Terms* (metaclasses) of *Dependency* and are used for the same purpose.

We will use *Derivation* relation (which shows as «derive» on the diagrams) in this Deskbook.

The easiest way to create the derived requirements is on requirements diagrams[16]. The next several figures show system level requirements and the requirements derived from them.

[16] Unless your requirements are being held in a DOORS NG respository; in that case, you'll have to do the derivation work in the DOORS NG tool.

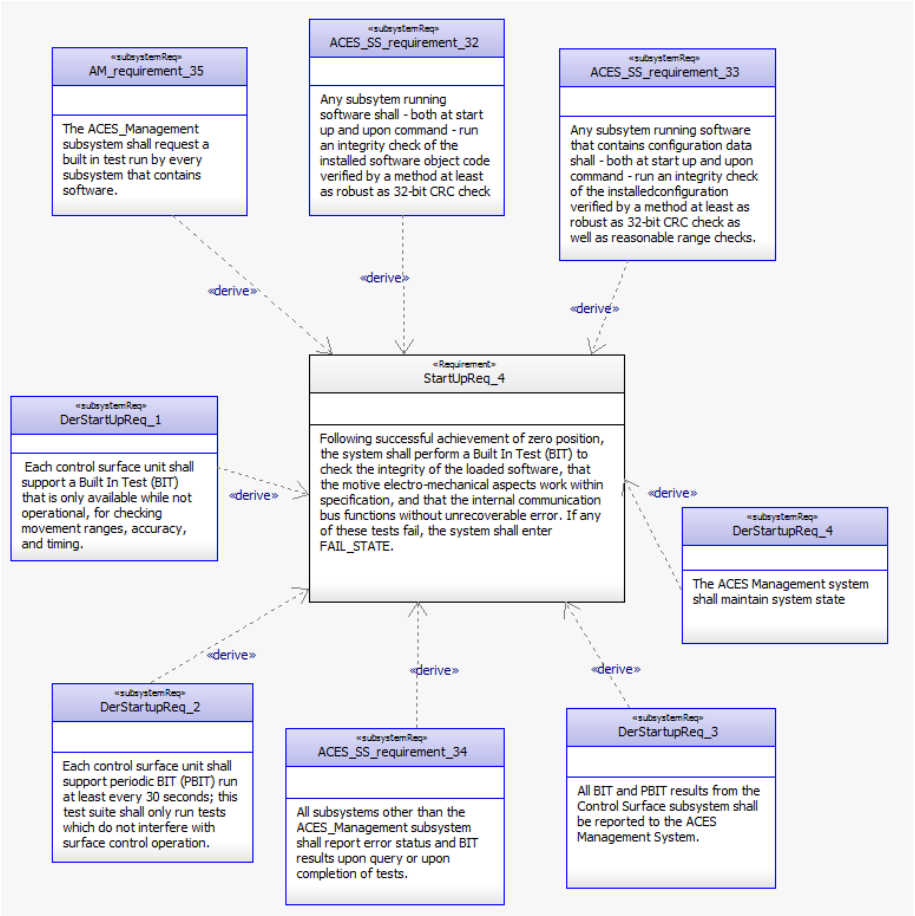


Figure 169: Derived Requirements

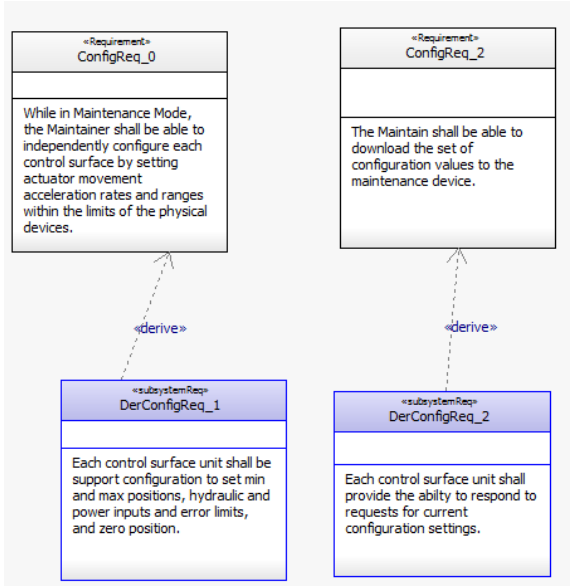


Figure 170: Derived Requirements

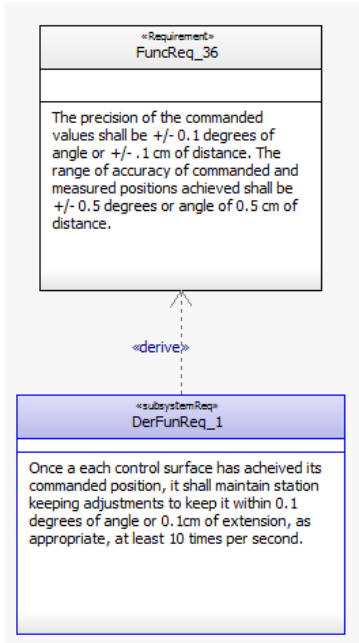


Figure 171: Derived Requirements

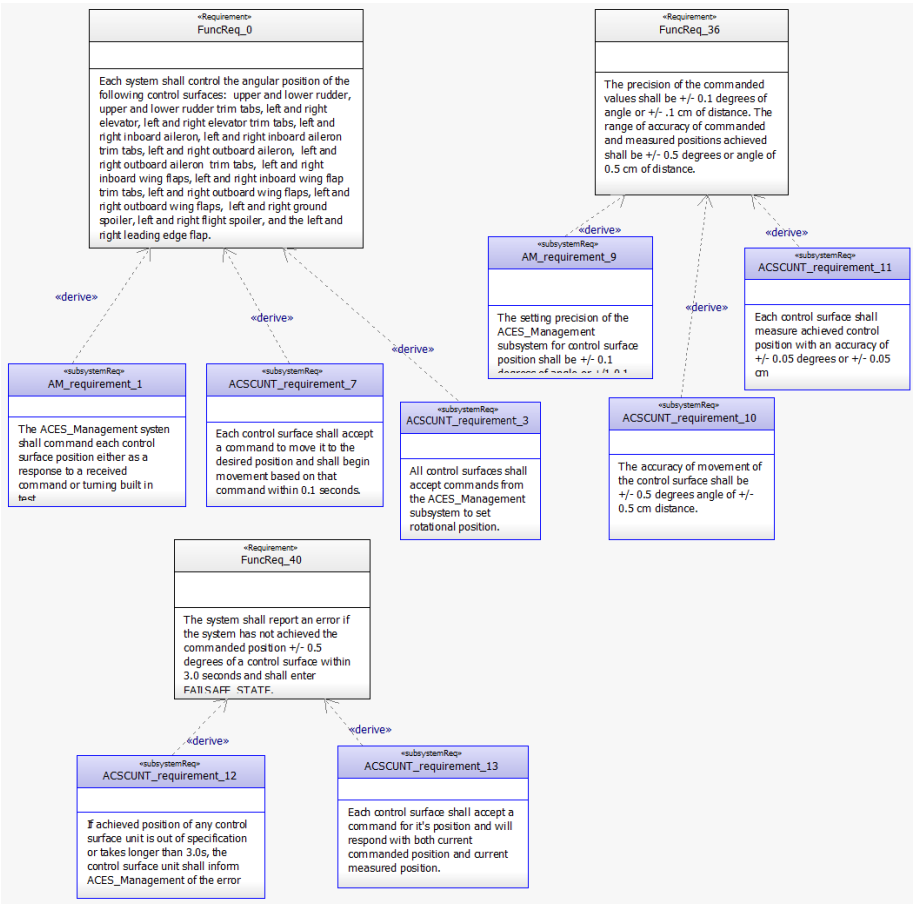


Figure 172: Derived Requirements

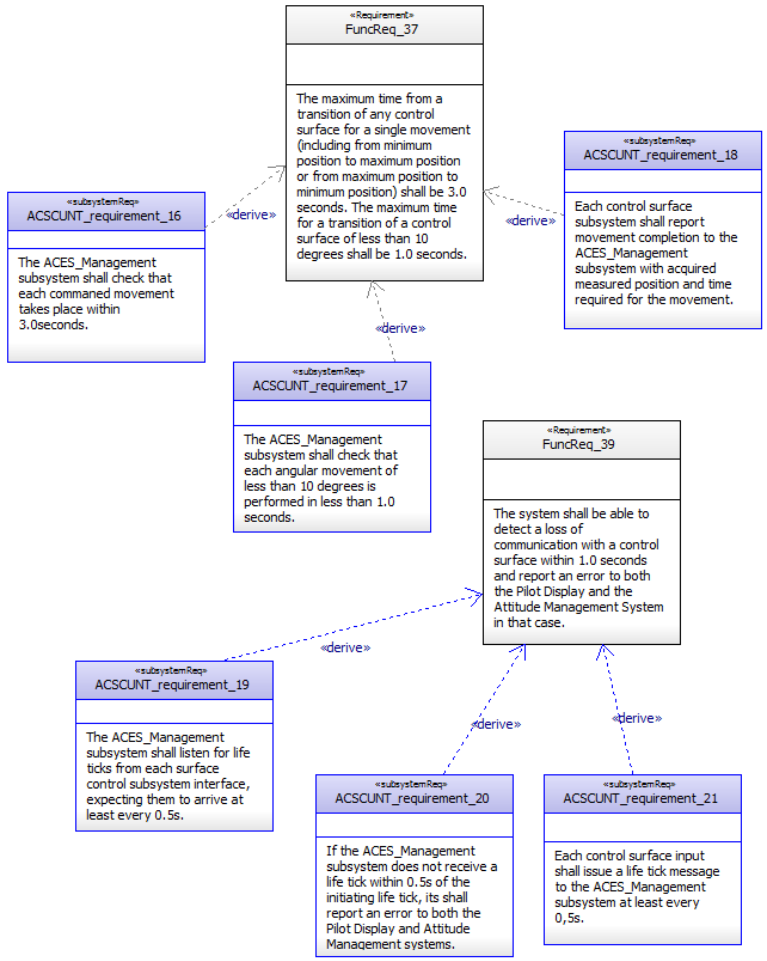


Figure 173: Derived Requirements

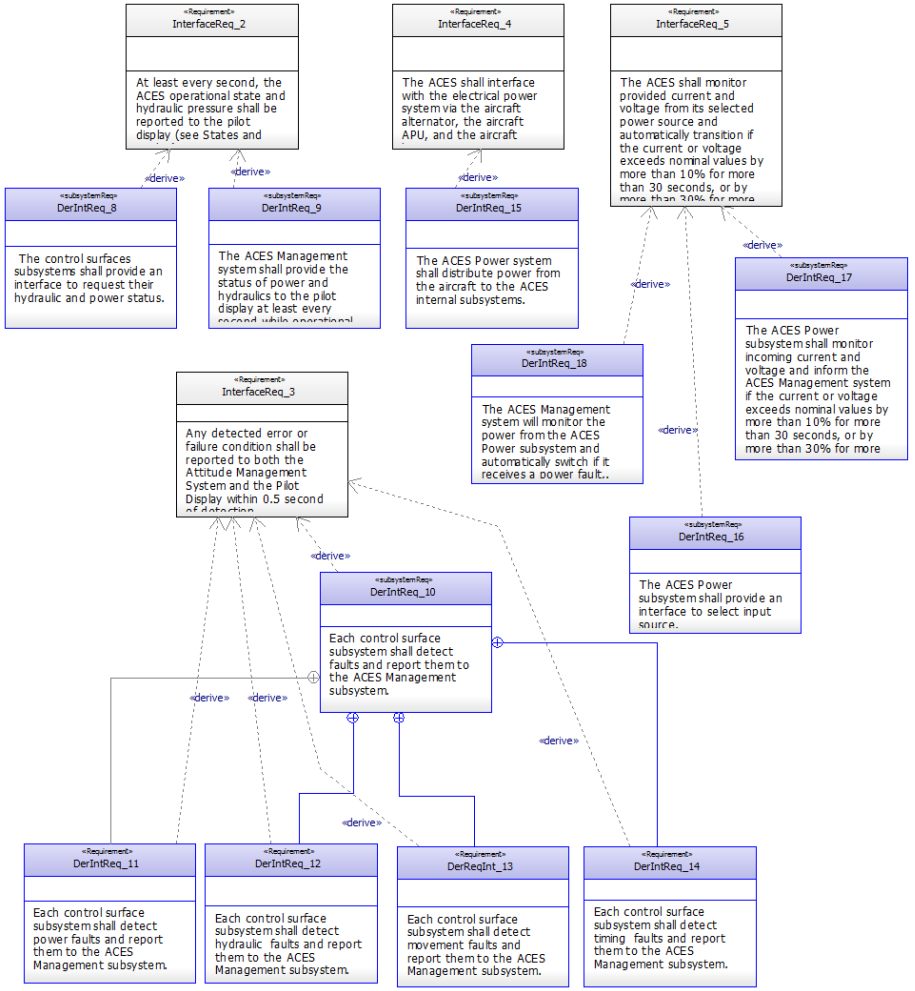


Figure 174: Derived Requirements

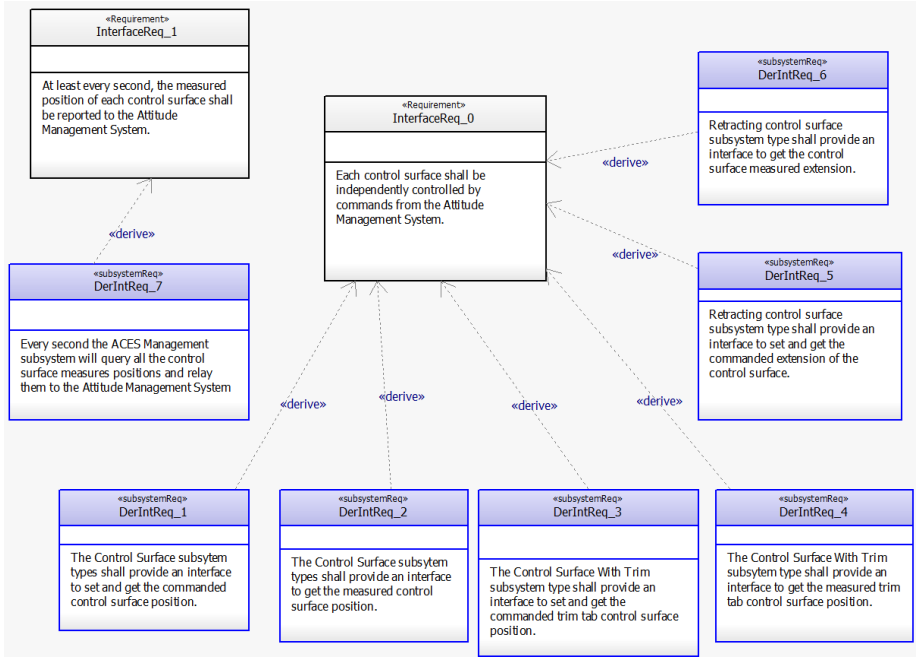


Figure 175: Derived Requirements

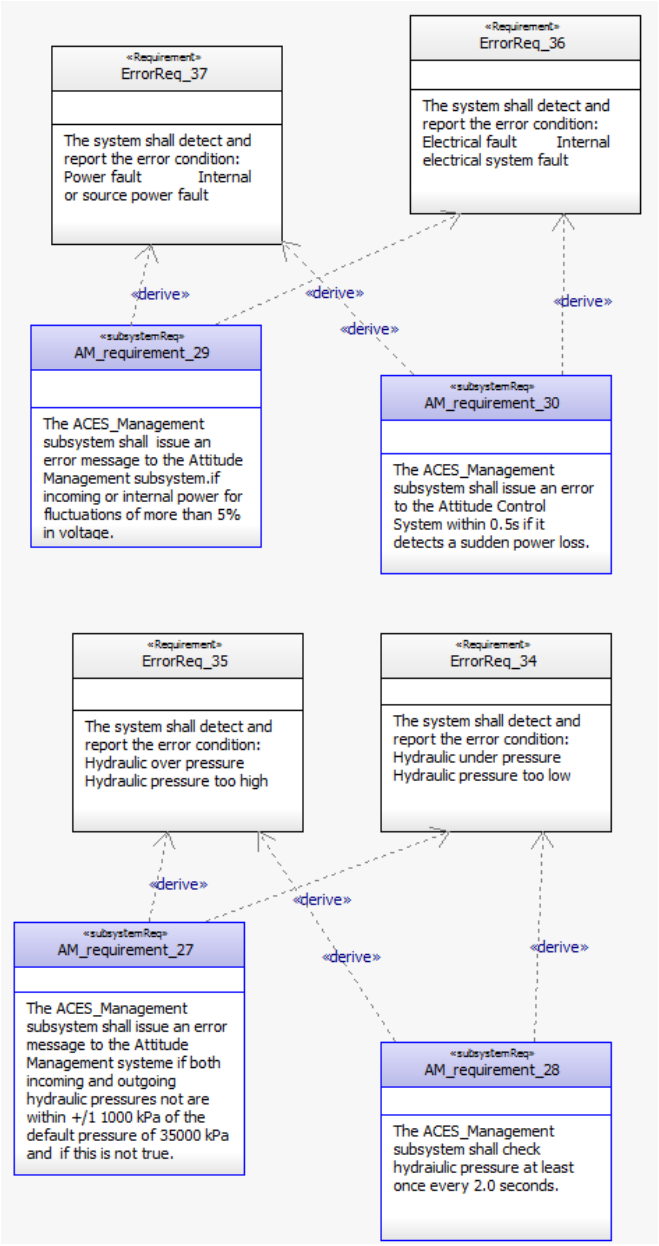


Figure 176: Derived Requirements

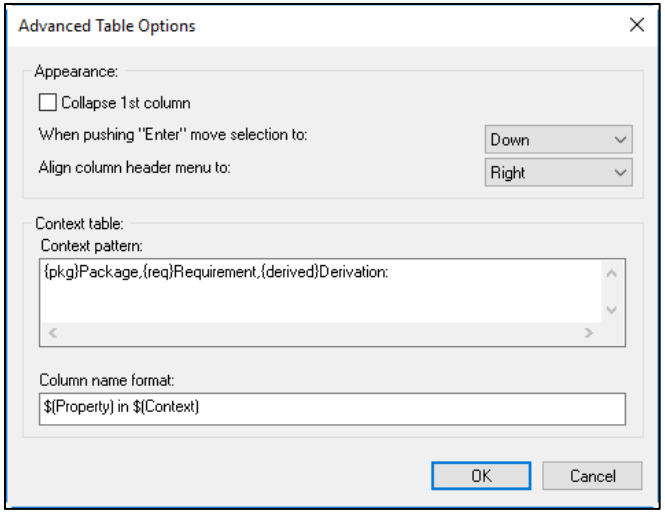
Showing the Derived Requirements

The derivations are best created diagrammatically, but they are perhaps best viewed in tabular format. To do this I first create a table layout providing the information I want, and then create a table view from that layout.

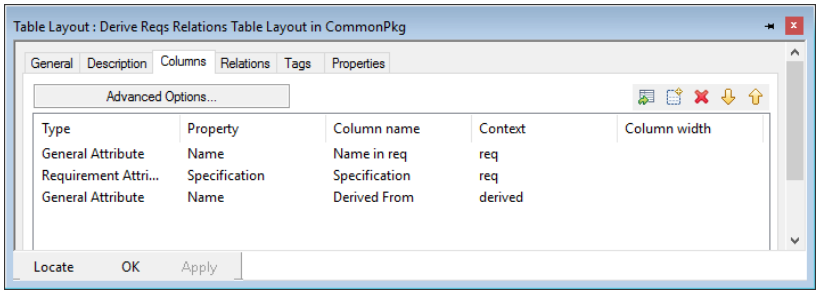
Creating the Table Layout

The table view we want includes the name of the derived requirement, the text of its specification, and the name of the requirement from which it is derived. To do this we'll create a new table layout using context patterns.

- Right click on the **CommonPkg** in the browser and select *Add New > View and Layouts > TableLayout*.
- Name this table layout **Derive Reqs Relations Table Layout**.
- Click on the *Columns* pane of the *Features* dialog
- Click on the *Advanced Options* button
- Add the following context pattern



- Add the following column definitions on the Columns pane



- Click on *OK*.

Creating the Table View

Creating a table view from this layout is easy.

- Right click on the table layout just created
- Select *Create View*
- Name this view **Derive Requirements Relations**
- The scope can be the entire model or limited to the **RequirementsAnalysisPkg**.

That table is shown below in Table 2.

Found 52 elements

Name in req	Specification	Derived From
ACES_SS_requirement_32	Any subsytem running software shall - both at start up and upon command - run an integrity check of the installed software object code verified by a method at least as robust as 32-bit CRC check	StartUpReq_4
ACES_SS_requirement_33	Any subsytem running software that contains configuration data shall - both at start up and upon command - run an integrity check of the installedconfiguration verified by a method at least as robust as 32-bit CRC check as well as reasonable range checks.	StartUpReq_4
ACES_SS_requirement_34	All subsystems other than the ACES_Management subsystem shall report error status and BIT results upon query or upon completion of tests.	StartUpReq_4
ACSCUNT_requirement_10	The accuracy of movement of the control surface shall be +/- 0.5 degrees angle of +/- 0.5 cm distance.	FuncReq_36
ACSCUNT_requirement_11	Each control surface shall measure achieved control position with an accuracy of +/- 0.05 degrees or +/- 0.05 cm	FuncReq_36
ACSCUNT_requirement_12	If achieved position of any control surface unit is out of specification or takes longer than 3.0s, the control surface unit shall inform ACES_Management of the error	FuncReq_40
ACSCUNT_requirement_13	Each control surface shall accept a command for it's position and will respond with both current commanded position and current measured position.	FuncReq_40
ACSCUNT_requirement_16	The ACES_Management subsystem shall check that each commaned movement takes place within 3.0seconds.	FuncReq_37
ACSCUNT_requirement_17	The ACES_Management subsystem shall check that each angular movement of less than 10 degrees is performed in less than 1.0 seconds.	FuncReq_37
ACSCUNT_requirement_18	Each control surface subsystem shall report movement completion to the ACES_Management subsystem with acquired measured position and time required for the movement.	FuncReq_37
ACSCUNT_requirement_19	The ACES_Management subsystem shall listen for life ticks from each surface control subsystem interface, expecting them to arrive at least every 0.5s.	FuncReq_39
ACSCUNT_requirement_20	If the ACES_Management subsystem does not receive a life tick within 0.5s of the initiating life tick, its shall report an error to both the Pilot Display and Attitude Management systems.	FuncReq_39
ACSCUNT_requirement_21	Each control surface input shall issue a life tick message to the ACES_Management subsystem at least every 0,5s.	FuncReq_39
ACSCUNT_requirement_24	Each control surface unit instance shall have a unique identifier which shall be used to in messages to the ACES_Management subsystem.	InterfaceReq_0
ACSCUNT_requirement_25	Each control surface unit shall have, as persistent configuration data, low and high movement limits, required measurement accuracy, and movement time	FuncReq_0
ACSCUNT_requirement_26	Each surface control unit instance shall report an error to the ACES_Management subsystem if the result of a commanded movement is out of specification either in accuracy or timing.	FuncReq_36
ACSCUNT_requirement_3	All control surfaces shall accept commands from the ACES_Management subsystem to set rotational position.	FuncReq_0
ACSCUNT_requirement_7	Each control surface shall accept a command to move it to the desired	FuncReq_0

Table 2: Derived Requirements Table (partial)

The complete table, formatted in Word, is shown in Section 13.1.

This table provides a useful view because it shows the derived requirements,their specifications, and from whence they came.

The basic rule of requirements traceability to the subsystem is that each requirement that traces to a use case must be allocated to a subsystem UNLESS it is decomposed into derived requirements. That means we need a way to easily identify those system requirements that are decomposed. For

this reason, the Harmony SE Profile contains the **«DecomposedRequirement»** stereotype. This stereotype applies only to requirements and has the tag **hasDerivedRequirements** (which defaults to TRUE) to mark such requirements.

Now go through all the requirements in the right hand column of Table 2 and apply the **«DecomposedRequirement»** stereotype.

9.2.2 Performing the allocation of requirements

All system requirements traced to use cases must either be directly allocated to the subsystems or have the requirements derived from it so allocated. It is also recommended, as previously mentioned, to trace to system features modeling the use cases and architecture – including event receptions, operations (system functions), value properties, types, states, transitions, actions (also system functions), subsystems and their features, and relations. In this step, we will perform the allocation to the subsystems.

Similar to the creation of the derived requirements, the allocation of requirements to subsystems can be done diagrammatically or in matrices.

To do the allocation diagrammatically, for each use case, create at least one requirements diagram in the **DesignSynthesisPkg > ArchitecturalDesignPkg** package. There may be more than one if you have many requirements for the use case. Next, on each such diagram, drag the requirements allocated to this use case onto this diagram. Finally, drag the set of subsystem blocks onto the diagram and start adding the *Allocate* relations from the subsystem to the appropriate requirements. At the end, there should be no requirement traced to a use case that is not also allocated to a subsystem unless it is decomposed into derived subsystem level requirements. Figure 177 shows an example.

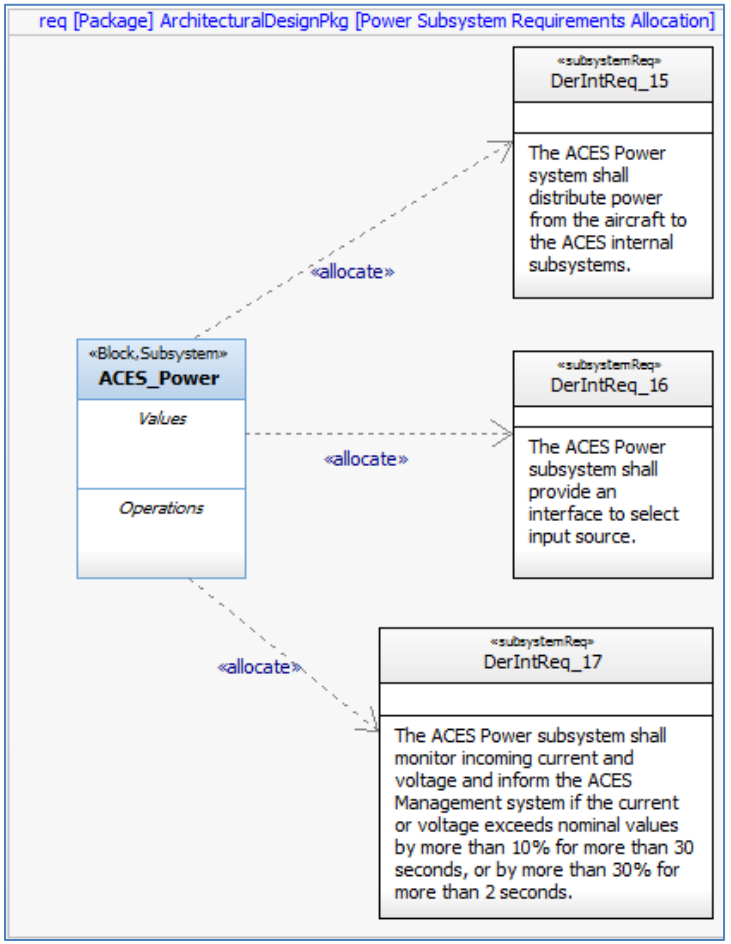
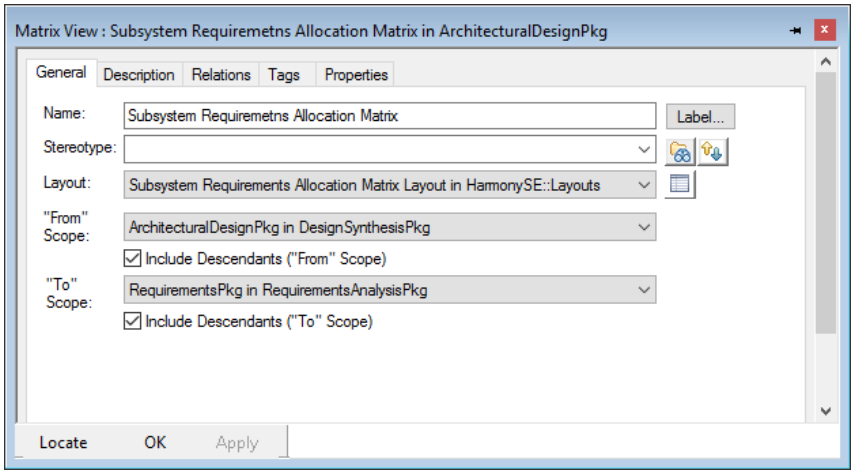


Figure 177: Allocation of Requirements to Power Subsystem

Using the Block-Requirement Allocation Matrix

In the *DesignSynthesisPkg > ArchitecturalDesignPkg* package add a *Matrix View* and name it **Subsystem Requirements Allocation Matrix**. Open its *Features* dialog and set its layout to the **Subsystem Requirements Allocation Matrix Layout** provided by the Harmony SE Profile. Set the *From Scope* to the **ArchitecturalDesignPkg** and the *To Scope* to the **RequirementsPkg**.



Click on *Ok*. Now you can double click on the matrix view in the browser and open it up. I recommend you click on *Switch Rows and Columns* in the Matrix toolbar because there are many more requirements than there are subsystems.

A portion of this matrix (with rows and columns switched) is shown below:

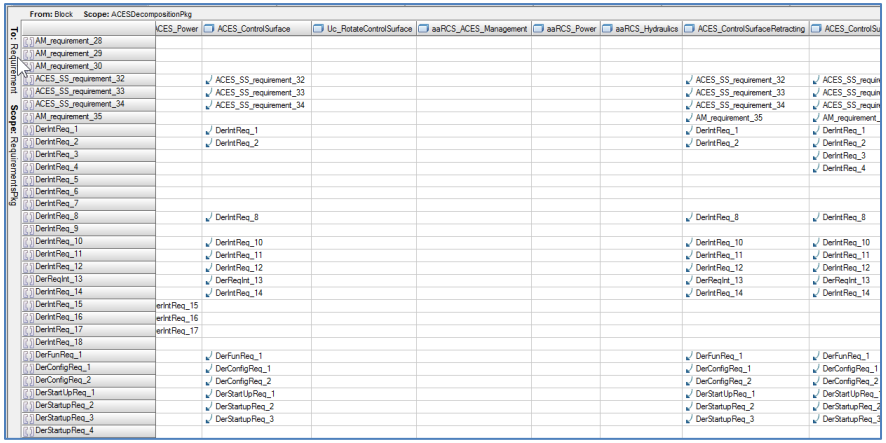


Figure 178: Portion of the Subsystem Requirements Allocation Matrix

Now you can work in this matrix view to set the allocation relations by selecting one or more cells, right click and select *Add New > Allocation* to populate the cell.

Review the allocation matrix

Once this is done, walk through the matrix to look at all the rows (assuming you switched rows and columns n the view, otherwise look for empty columns). Each row corresponds to a requirement; is that requirement marked with the **«DecomposedRequirement»** stereotype? If not, be sure to allocate it (or decompose it into derived requirements)[17].

To show the allocations here, we'll build up a subsystem-requirements allocation table.

The **Subsystem Requirements Table Layout** uses the following context pattern and columns:

[17] Tip: This can also be done by exporting the matrix to a CSV file (using the export tool on the drawing toolbar), loading it in Microsoft Excel, and using the COUNTA function to count the non-empty cells in the columns.

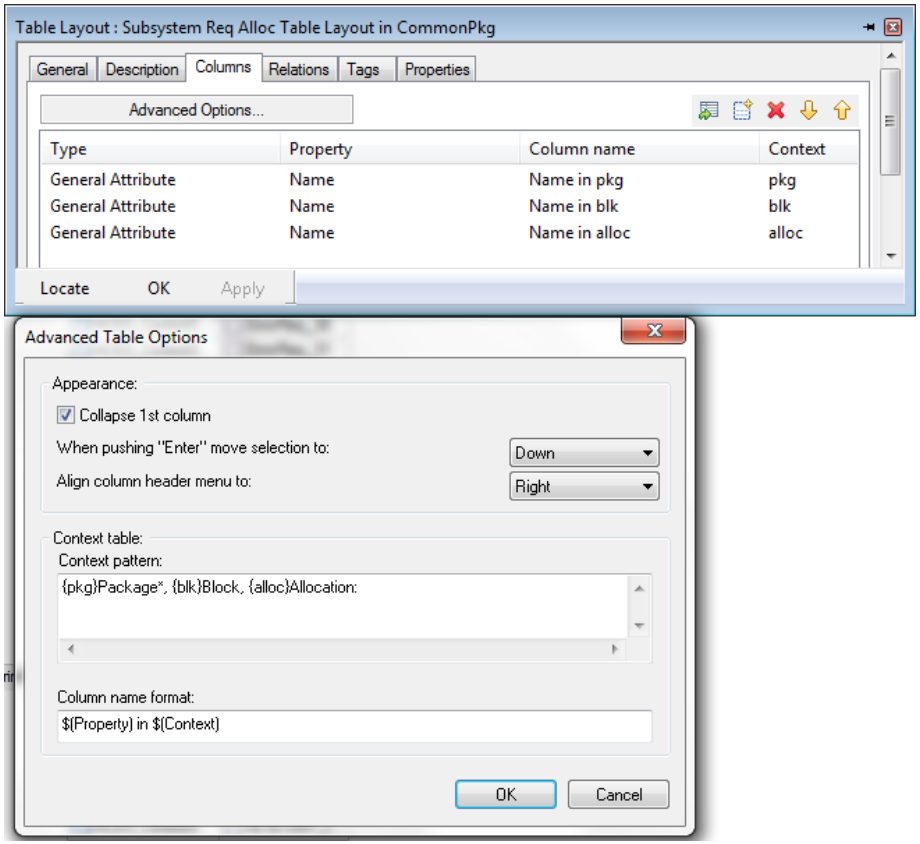


Figure 179: Subsystem Requirement Allocation Table Layout options

A portion of this table is shown below:

Name in pkg	Name in blk	Name in alloc
ACESDecompositionPkg		
ACES_Control_SurfacePkg		
ACES_Control_Surface_RetractingPkg		
ACES_Control_Surface_With_TrimPkg		
ACES_HydraulicsPkg		
	ACES_Hydraulics	SafetyReq_001
	ACES_Hydraulics	SafetyReq_002
	ACES_Hydraulics	SafetyReq_004
	ACES_Hydraulics	SafetyReq_005
	ACES_Hydraulics	SafetyReq_390197
	ACES_Hydraulics	Safety_Req_390198
	ACES_Hydraulics	Safety_Req_390200
	ACES_Hydraulics	Safety_Req_390201
	ACES_Hydraulics	Safety_Req_390209
ACES_ManagementPkg		
	ACES_Management	ACES_SS_requirement_32
	ACES_Management	ACES_SS_requirement_33
	ACES_Management	ACES_SS_requirement_34
	ACES_Management	ACSCUNT_requirement_12
	ACES_Management	ACSCUNT_requirement_13
	ACES_Management	ACSCUNT_requirement_16
	ACES_Management	ACSCUNT_requirement_17
	ACES_Management	ACSCUNT_requirement_18
	ACES_Management	ACSCUNT_requirement_19
	ACES_Management	ACSCUNT_requirement_20
	ACES_Management	ACSCUNT_requirement_21
	ACES_Management	ACSCUNT_requirement_24
	ACES_Management	ACSCUNT_requirement_26
	ACES_Management	ACSCUNT_requirement_3
	ACES_Management	AM_requirement_1
	ACES_Management	AM_requirement_27
	ACES_Management	AM_requirement_28
	ACES_Management	AM_requirement_29
	ACES_Management	AM_requirement_30
	ACES_Management	AM_requirement_35
	ACES_Management	AM_requirement_4
	ACES_Management	AM_requirement_6
	ACES_Management	AM_requirement_9
	ACES_Management	ConfigReq_1
	ACES_Management	ConfigReq_3
	ACES_Management	DerIntReq_11
	ACES_Management	DerIntReq_12
	ACES_Management	DerIntReq_14
	ACES_Management	DerIntReq_16
	ACES_Management	DerIntReq_17
	ACES_Management	DerIntReq_18
	ACES_Management	DerIntReq_7
	ACES_Management	DerIntReq_9
	ACES_Management	DerReqInt_13
	ACES_Management	DerStartupReq_4
	ACES_Management	ErrorReq_0
	ACES_Management	ErrorReq_1
	ACES_Management	ErrorReq_10
	ACES_Management	ErrorReq_11
	ACES_Management	ErrorReq_12
	ACES_Management	ErrorReq_13
	ACES_Management	ErrorReq_14
	ACES_Management	ErrorReq_15
	ACES_Management	ErrorReq_16

Table 3: Subsystem Requirement Allocation Table (partial)

See the entire table, formatted in Word, in Section 13.2.

9.3 Allocate Use Cases to Subsystems

For subsystems with few requirements, it is not necessary to create subsystem level use cases. However, most subsystem have a large number of requirements, and the same benefits that use cases provide for organizing and managing system requirements also apply at the subsystem level.

It is important to note that system use cases are almost never allocated to a single subsystem. They must always be decomposed into smaller "subsystem-level" use cases that can be so allocated.

There are two different approaches to allocating use cases to subsystems (Figure 11, page 20). The "top-down" approach works at the use case level, using the «include» relation as a kind of logical containment of subsystem level use cases. These subsystem level use cases are then either allocated directly to subsystems or further decomposed until they can be. The "bottom up" approach allocates actions (system functions) from the use case white box activity diagram or sequence diagrams to subsystems that represent the subsystems. These derived diagrams are called *white-box* diagrams because they expose the subsystem architecture as either swim lanes in activity diagrams or lifelines in sequence diagrams.

In general, larger systems are more easily developed with the top down approach while smaller systems are more easily developed with the bottom up approach. Nevertheless, both workflows are effective, and which approach you take is, to some degree, a matter of personal preference.

In our pedagogical approach in this Deskbook, we'll taken two approaches to analyze use cases. The first used the activity-based approach. That approach lends itself well to the bottom up approach. We will use the bottom-up approach for the analysis of the **Start Up** use case and the top-down approach for the **Control Air Surfaces** use case.

Again, if you only allocated a few requirements to a subsystem (say, less than 20), it may not make any sense to define subsystem level use cases.

9.3.1 Bottom-Up Approach: Start Up Use Case

This is an approach favored by many systems engineers. In this case, we will create white box sequence diagrams (sequence diagrams that include subsystems as lifelines) to perform the allocations rather than create white box activity diagrams.

9.3.1.1 But what about White Box Activity Diagrams?

The use of white box activity diagrams to show and aid in the allocation of system properties was a prominent feature of "Harmony Classic." While it has some positive aspects (notably, it's highly visual), it has some serious drawbacks, which is why we no longer recommend it. Specifically:

- In Harmony Classic, the state machine is the "source of truth".
  The activity diagram shows the primary flows not not all the detail. Rarely are all requirements (especially edge cases and exception handling) are represented in the activity diagram. To use white box activity diagrams for allocation, you would have to add these missing functions and flows to the activity diagram so that all use case requirements are represented.
- You don't verify the activity diagram.
  You verify the state machine so the state machine is the "source of truth". There is both additional work and a possibility of introducing errors by manually backfilling the activity diagram as you discover requirements issues during the development and verification of the state machine.
  The reason to build the state machine is to create a precise and verifiable statement of the requirements for the use case. If you prefer to work only with the activity diagram (a reasonable thing, after all), then you would be better served using the full "Flow Based Approach" using full, executable activity diagrams as shown on the left side of Figure 4.
- You have potentially conflicting "sources of truth".
  If both the activity diagram and state machine are treated as equivalent sources of truth, if they are in conflict, which is deemed to be correct?

Having said that, the discussion below shows you what it would look like, if you to proceed in this fashion.

The allocation of these actions is done by duplicating the use case activity diagram, adding swim lanes representing the subsystems, and dragging the actions to the appropriate swim lane. The SE Toolkit will help us along the way automating aspects of different steps.

- Right click on the activity diagram *FunctionalAnalysisPkg > StartUpPkg > Use Cases > StartUp > Activity Views > Start UpBlackBoxView* and select *SE-Toolkit > Create White Box Activity View*. This action create a copy of the black box activity diagram with which we can work.
- Add a swimlane from to the newly *created Start UpWhiteBoxView > activity_0*, Start *UpWhiteBoxView > RangeSurfaceTest*, and *Start UpWhiteBoxView > PerformBIT* activity diagrams.
    - In the main activity diagram (**activity_0**), leave the two call activities (**RangeSurfaceTest** and **PerformBIT**) outside the swimlane frame, since their allocations will be shown on their respective diagrams.

Each of these diagrams with have different swim lanes because they use different subsystems.

Activity_0 activity diagram is the overall behavior

- To this diagram add 2 swim lanes, one for the **ACES_Management** subsystem and the other for the **ACES_Power subsystem**. To associate the swim lane with the subsystem, you can drag the subsystem block to the *top* of the swimlane or you can go through the *Features* dialog:
    - Double click on the swim lane to get the *Features* dialog.
    - In the *Represents* drop down list, use select to navigate the the appropriate subsystem, noting that the previous toolkit action moved the subsystem blocks into their own nested packages:

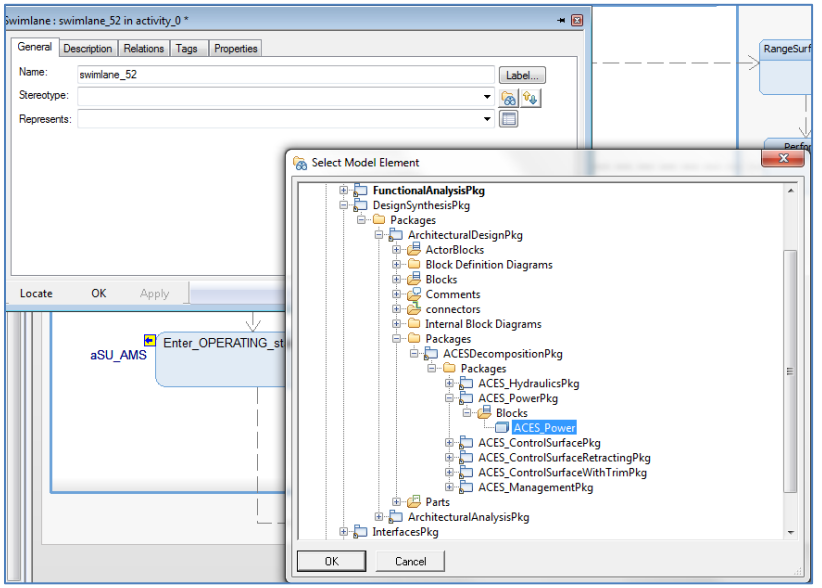


Figure 180: Associating swim lanes with subsystems

- Move the **Select_Battery_As_Source** action to the **ACES_Power** swimlane and all the other actions in the frame to the **ACES_Management** swimlane.

The resulting diagram should look like Figure 181.

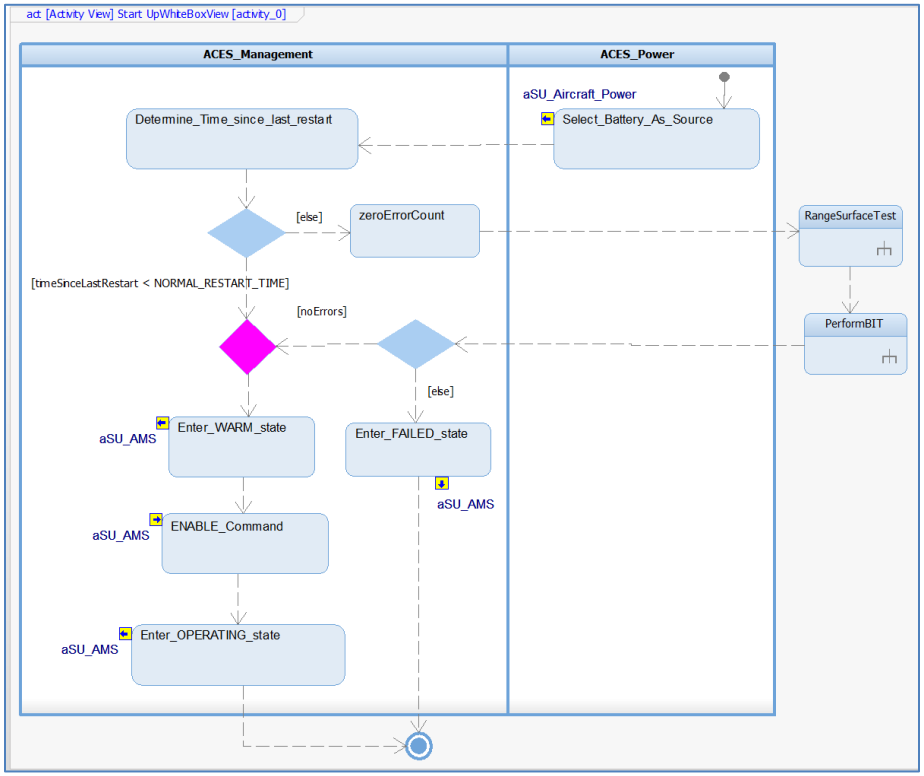


Figure 181: White Box Activity Diagram for Start Up Use Case (main diagram)

We must repeat this allocation for the activity diagrams referenced by the call activities in Figure 181.

For the **Range Surface Test** activity, you can see that we derived the need for some additional actions during the allocation process. This is includes actions such as **Goto_Minimum_Position**, **Get_Position_and_Movement_Time**, and **Compare_Position_and_Timeliness**.

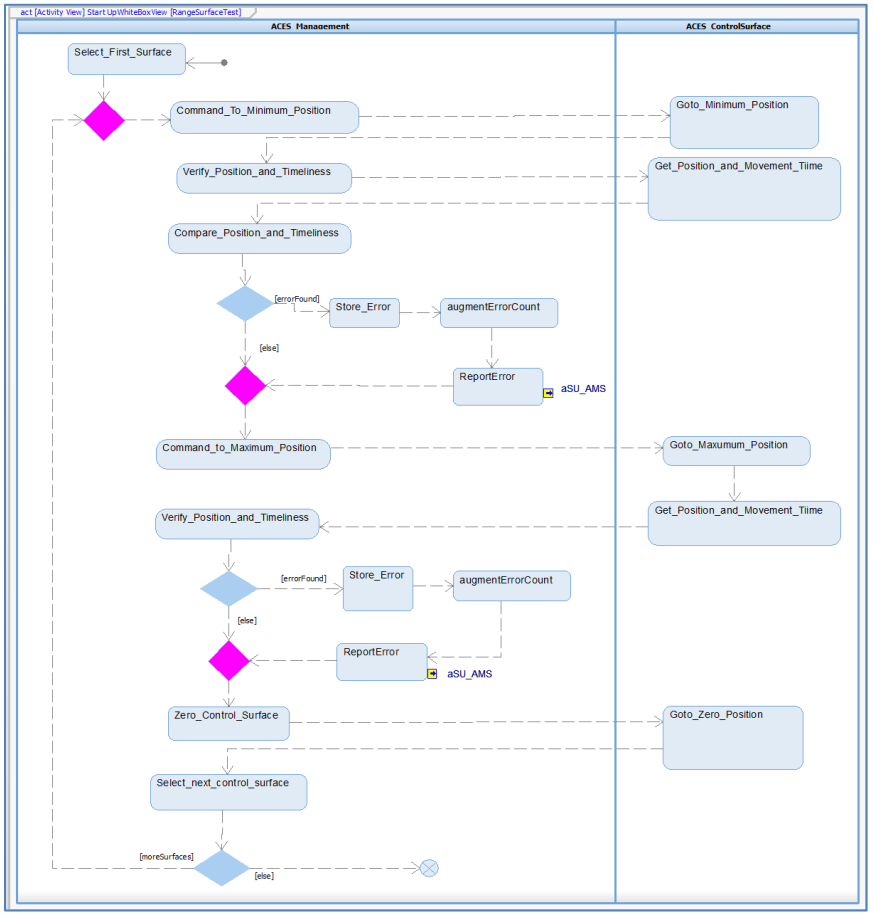


Figure 182: White Box Activity Diagram for Range Surface Test

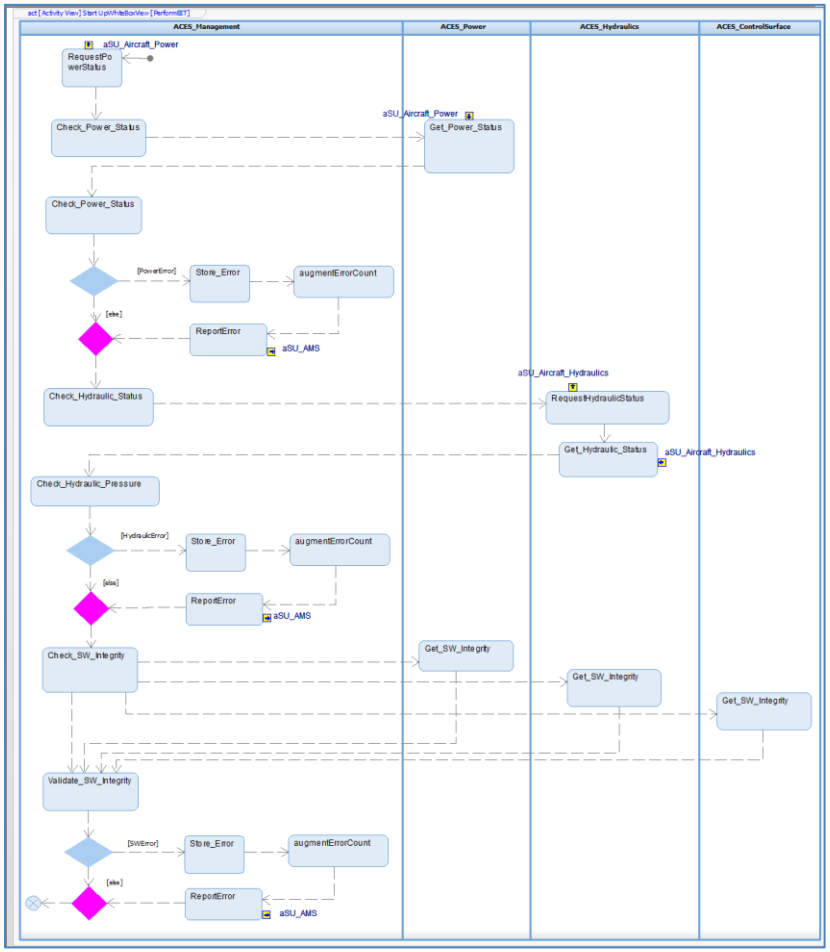


Figure 183: White Box Activity Diagram for Perform BIT

From here, you can proceed to use the SE Toolkit to create sequence diagrams, ports and interfaces, as discussed in the next section.

As you can see, this approach is straightforward. In Rhapsody 8.2 and later, placement of an action in a swim lane that represents a block creates an inferred allocation relationship.

9.3.1.2 Derivation of White Box Sequence Diagrams

The above section shows how you can use white box activity diagrams to do allocations. Nevertheless because of the concerns discussed in the previous section, we will perform this task using white box sequence diagrams instead.

The method to create and use white box scenarios is straight-forward.

- Create a package in **DesignSynthesisPkg > ArchitecturalDesignPkg** named **WBScenariosPkg**. Inside this package, add a package for every use case (to hold the white box versions of those scenarios), named ***<use case>*WBScenariosPkg**, such as **StartUpWBScenariosPkg** and **ControlAirSurfacesWBScenariosPkg**.
- For each use case in the included in the iteration:
  a. For each sequence diagram in the use case:
     i. Copy (not move) the sequence diagram to its appropriate package in the **DesignSynthesisPkg > ArchitecturalDesignPkg > WBScenariosPkg**. Dragging with the *control* key pressed is an easy way to do this.
     ii. Remove (not delete from model!) the comment describing the scenario from the diagram and replace it with a brand new one describing the white box version
     iii. Retarget each local use case actor block lifeline with the actual system actor (for example **aSU_Aircraft_Power** would be replaced with the **Aircraft_Power** actor). To do this, double click on the lifeline and in the *Realization* drop down list, select the appropriate actor (usually at the top of the list) and press *OK*.
     iv. Add the set of subsystems to the new sequence diagram (this can be done by right clicking on the sequence diagram and selecting *SE Toolkit > Add Subsystems*).
     v. For messages from the actors to the use case block: change target of messages to the appropriate subsystem

vi. From messages from the use case block to the actor: change the source of outgoing messages to one of the subsystems
vii. Elaborate the collaboration by adding messages among the subsystems, generally starting with the subsystem receiving the incoming actor message and terminating with subsystem sending the outgoing message to the actor
viii. Verify, via review, that the messages align with the allocations you made in Section 9.1.2.
ix. Once complete, then realize the messages on the sequence diagram by right clicking on the diagram and selecting *Auto Realize All Elements*.

By realizing the messages on the sequence diagrams, those messages will create operations or event receptions on those subsystems. These will serve as the basis for defining the ports and interfaces among the subsystems in the next step.

Let's do this for the sequence diagrams in the **Start Up** use case.

Creating the package for the white box sequence diagrams ends up with a structure that looks like this (note that I manually appended " WB" to the name of each copied sequence diagram):

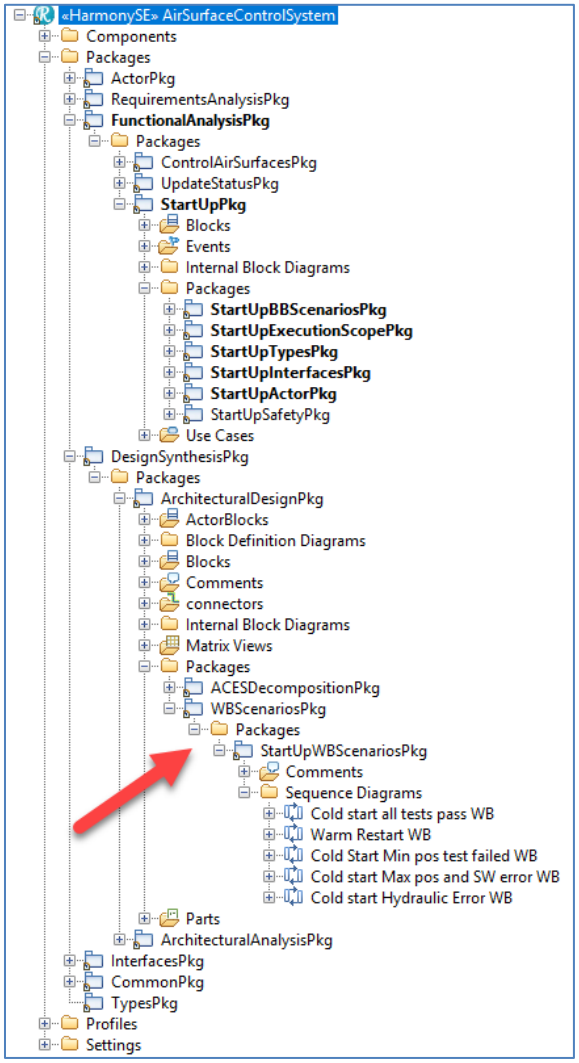


Figure 184: Start Up Use Case White Box Scenarios

As we move messages, the left or right arrow keys are useful for moving selected messages to the left or right lifelines.

A couple of guidelines for message refactoring in the white box sequence diagrams.

- "Messages to Self" can be either operation calls (synchronous) or events (asynchronous)
- Messages between subsystem should be events (asynchronous)
- It's ok to refine the messages so they make more sense in the context of defining services provided by subsystems. For example **Command_To_Minimum_Position(sID)** from the black box sequence diagram operation call might translate to a set of messages such as **Req_Minimum_Position(sID)** followed by **Command_To_Position(sID, pos)** and **Updated_Position(sID, pos, timing)** sent from the control surface back to the **ACES_Management** subsystem.
- It is important to remember that a life line on a sequence diagram corresponds to a singular instance at run-time. This is particularly relevant to our situation since we have 36 control surfaces. If you wanted to show the complete sequence for all surfaces, you'd have to show 36 different life lines for the control surface instances. We'll address that issue by only including a single lifeline in our scenarios generally.
- Once all messages are moved to the subsystems, then the now-unused use case life line may be removed.
- Update the (copied) diagram comment to reflect the white box content of the sequence diagram

In this section, we will create white box versions of all the non-animated sequence diagrams shown in Figure 184.

The first black box sequence diagram to be so transformed is Figure 63, "Generated Sequence diagram for warm restart". Note that the messages are moved to the subsystems. The subsystem lifelines are colored for ease of identification.

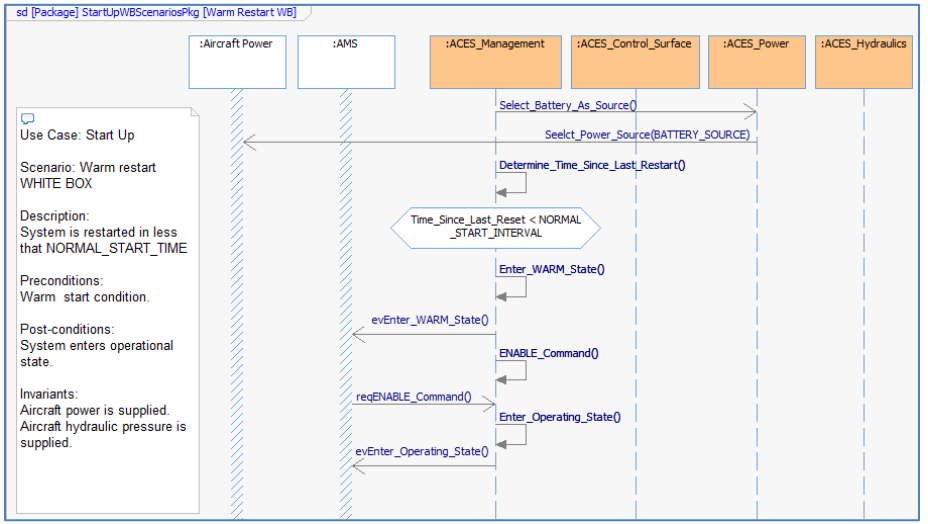


Figure 185: Start Up Use Case Scenario 1

The next figure is the white box version of Figure 65 "Cold Start All Tests Pass".

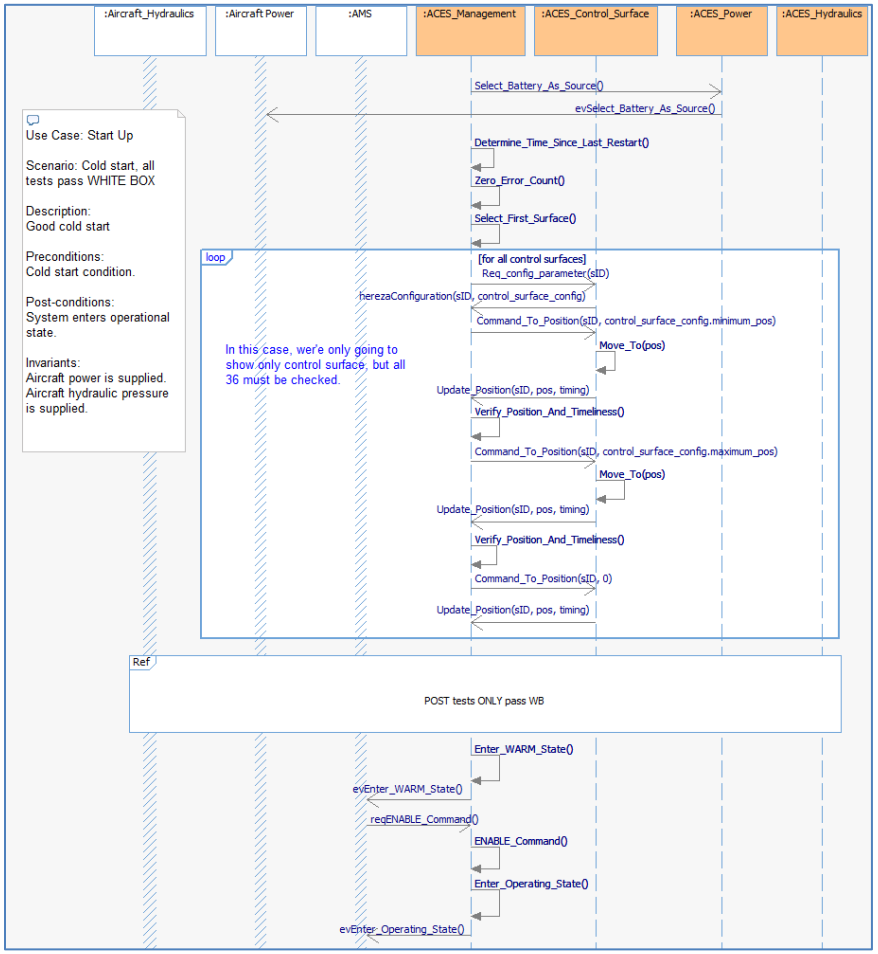


Figure 186: Start Up Use Case White Box Scenario 2

The referenced interaction fragment in Figure 186 is shown in Figure 187.

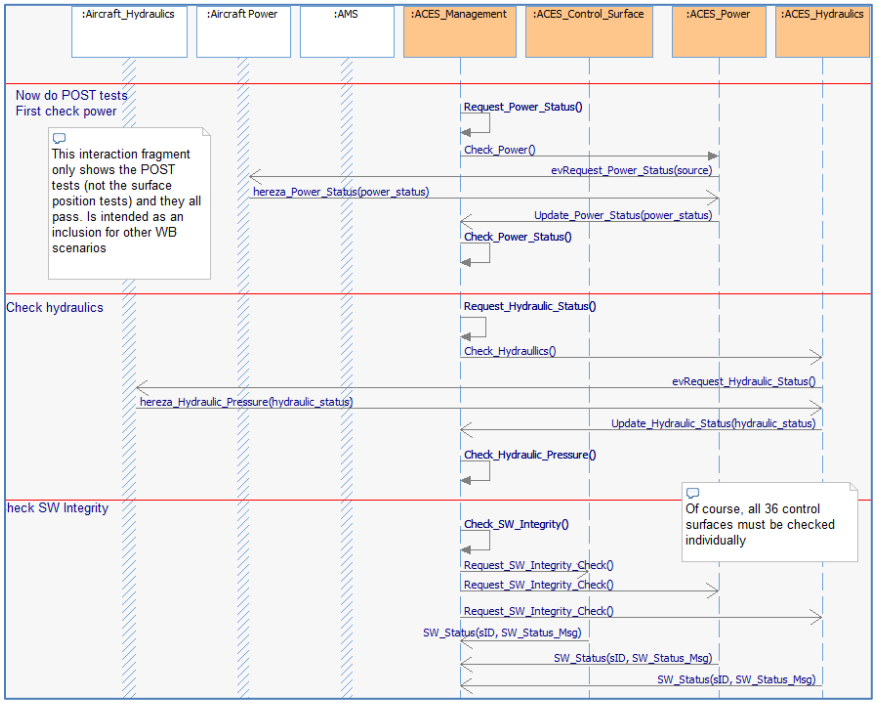


Figure 187: POST tests ONLY pass WB Interaction Fragment

Figure 188 shows a single error in the minimum position test. It is the white box equivalent of Figure 66.

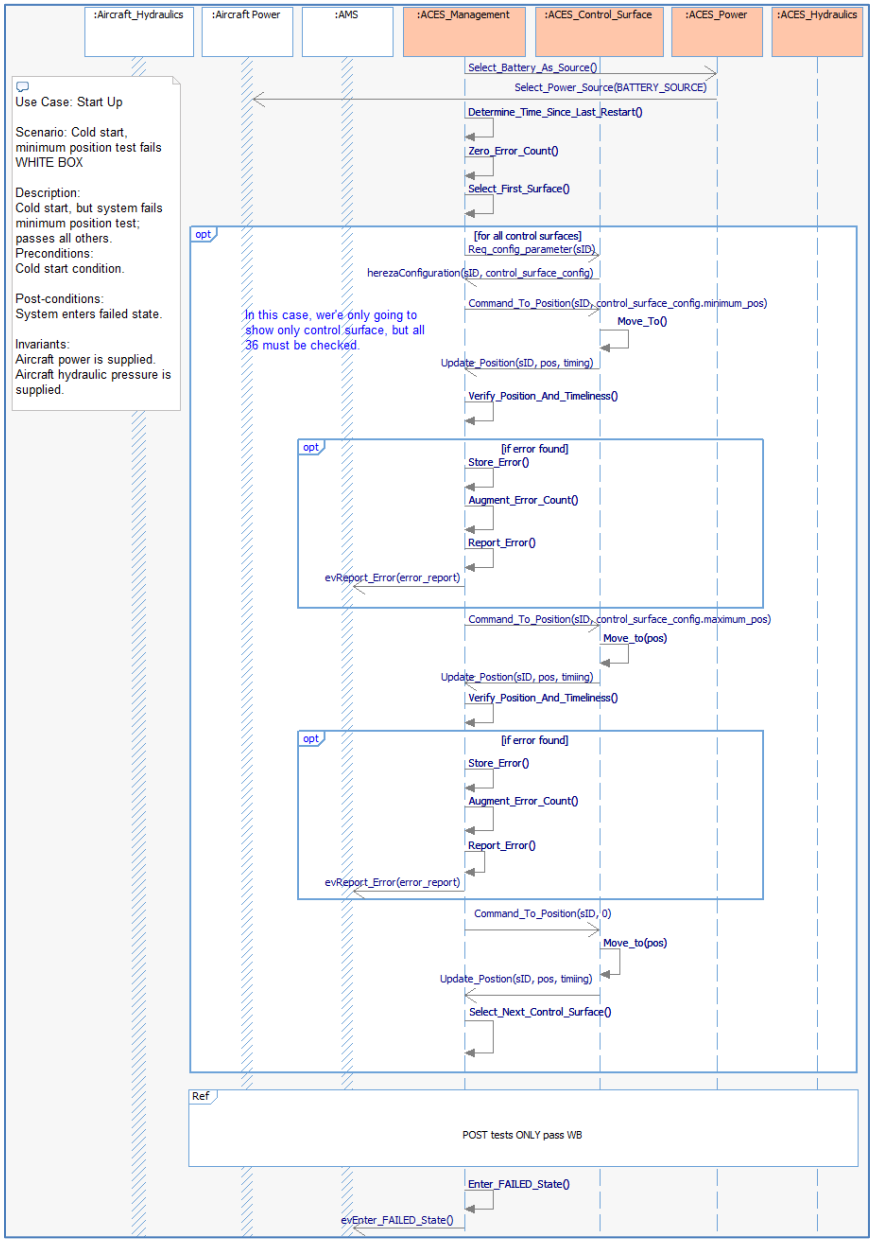


Figure 188: Start Up Use Case Scenario 3

The next figure (last in this sequence) shows multiple errors, including an error achieving the maximum position of a control surface and an error in the software component of the power system. Since we're using interaction operators to show multiple paths, it is very similar to the previous figure but refences a different interaction fragment.

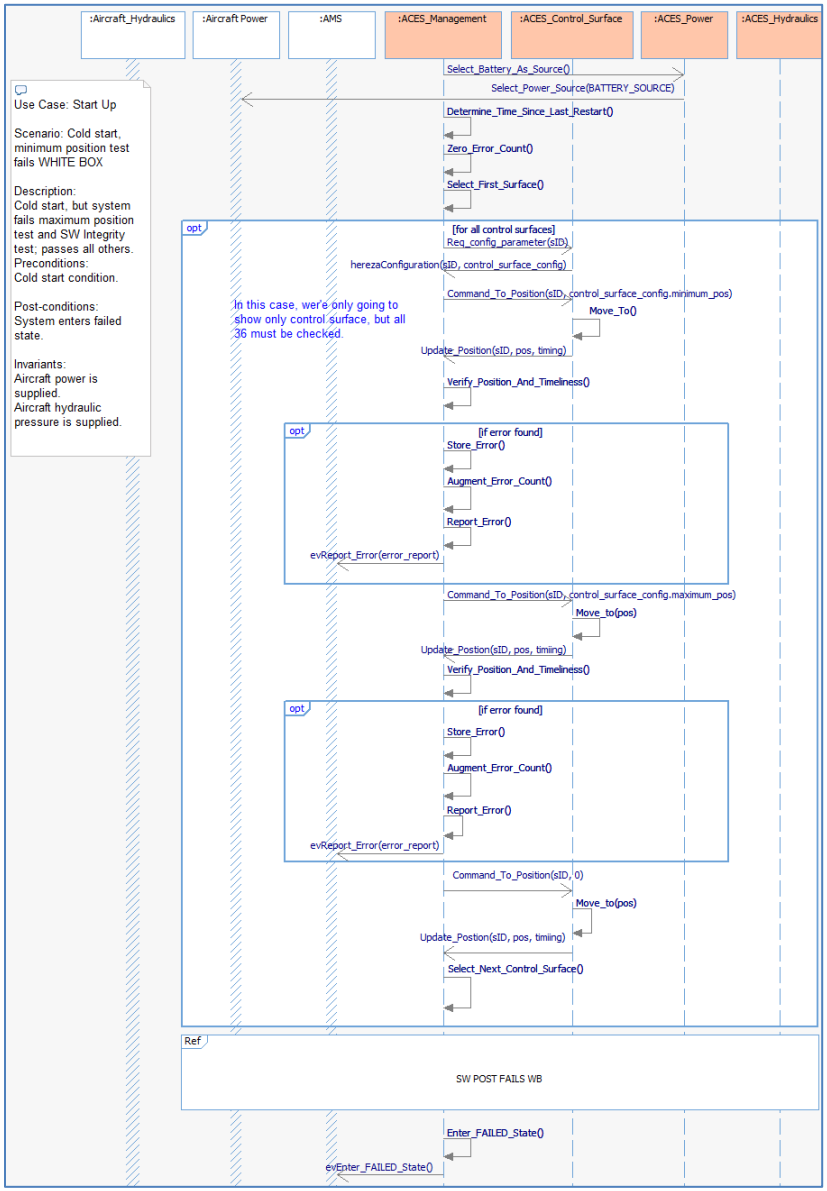


Figure 189: Start Up Use Case Scenario 4

The referenced interaction fragment shows the SW Integrity test for the Power Subsystem fail (Figure 190).

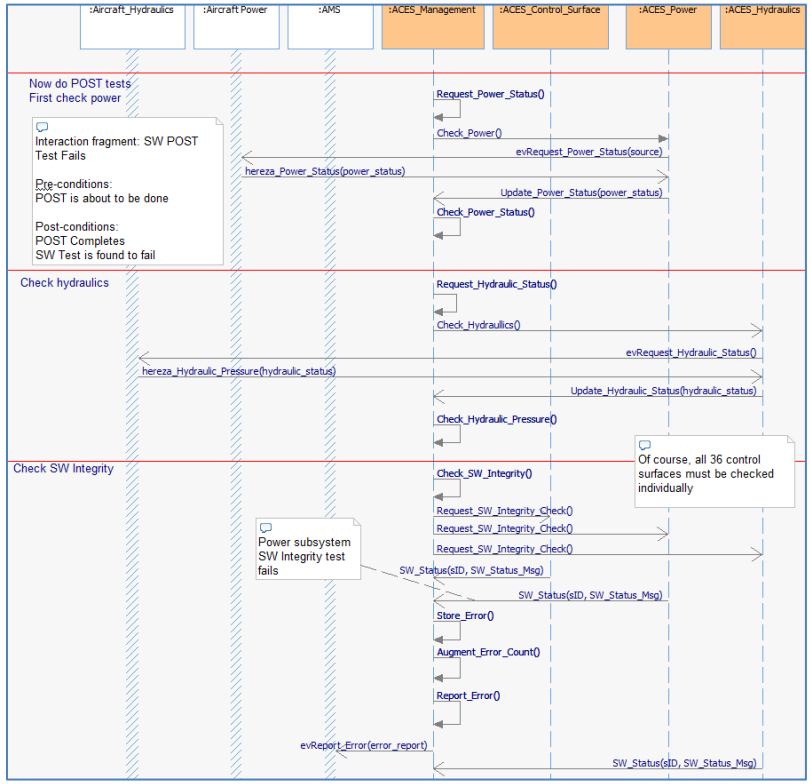


Figure 190: SW Integrity Test Fails Interaction Fragment

Once you have created all the diagrams, be sure to right click in each diagram and select *Auto Realize all elements*.

9.3.1.3 Define Subsystem Ports and Interfaces

The next step is to use the defined messages between the sbsystems and actors in the white box sequence diagrams to specify the interfaces.

- In the brower, right click on the **WBScenariosPkg**
- Select *SE-Tookit > Ports and Interfaces > Create Ports and Interfaces Recursive*

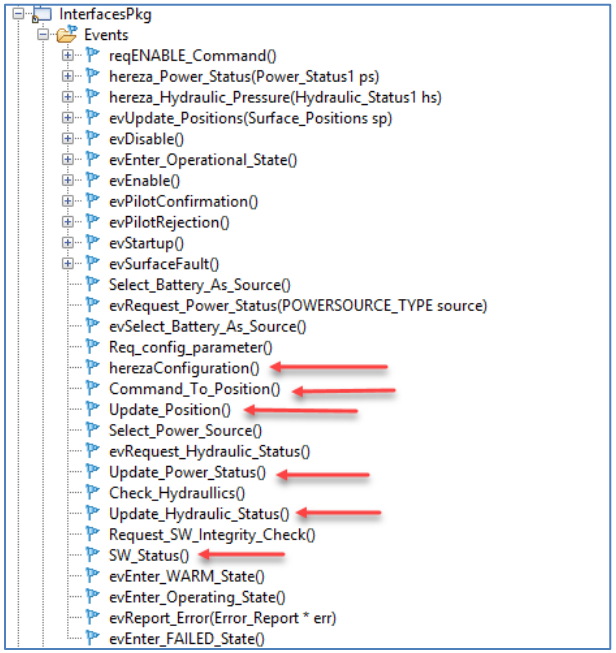
This step creates ports between communicating elements (subsystems and actors), creates interface blocks (if you're using the agile form of the tool,

otherwise it will create interfaces) and populates them with the events sent between them, as shown in Figure 189.

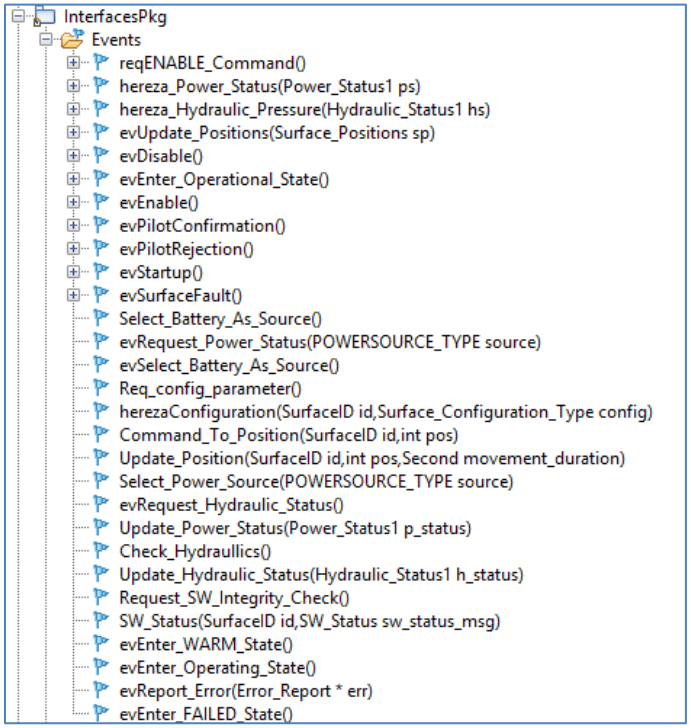
What the toolkit doesn't do is to copy in parameters into the messages, or at least for the most part. So editing the event receptions to ensure they have the proper parameters is a task that you must perform manually.

- Remove all «nonNormative» events (unless, for some reason, you decide to keep them in the specification to the subsystem)
- Walk through the events in the InterfacesPkg to be sure all the events have the proper parameters

The clones of the events from the functional analysis that had parameters with also have parameters; however, the events we added between the subsystems will not. Looking though the event list, I see the following events that must be updated to include parameter lists:

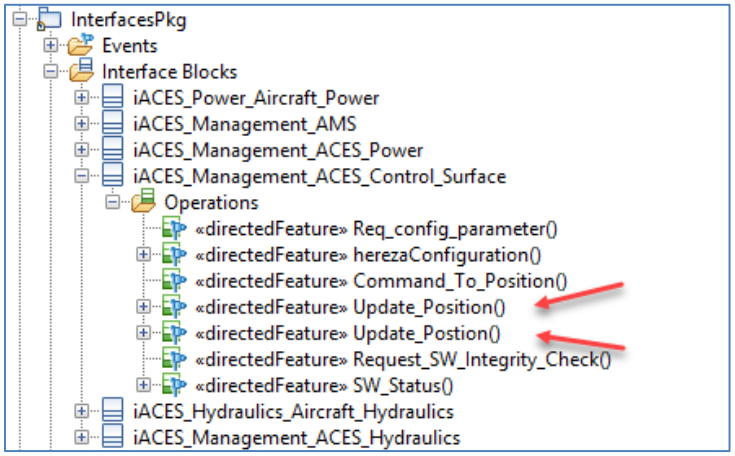


Adding the parameters should result in an event list like this:



Another thing you may face (I always seem to) is to find misspellings of the event names I want to send. What is **evConfiguration** in one sequence diagram might become **evConfiguriation** in another. Such misspellings or errors will need to be manually identified and repaired as well. At this point, you should

- walk thorough the operations in the interface blocks, looking for mistakes such as these:



Lastly, if you drew any *Messages* between lifelines on your sequence diagram that are not *Event Messages*, then the *Create Ports and Interfaces* tool will ignore those and not add them to the interface blocks.

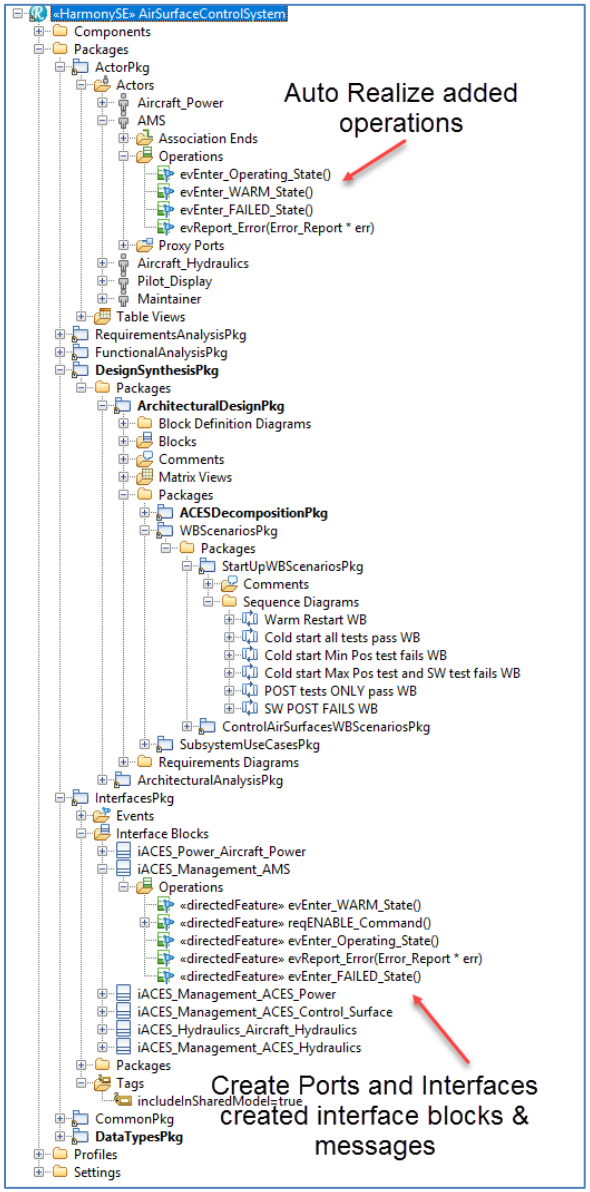


Figure 191: Create Ports and Interfaces Recursive outcome

9.3.1.4 Group Services Together into Use cases

By this point, we have:

- Identified subsytems
- Allocated operations and value properties to subsystems
- Allocated requirements to subsytems, including created subsystem-level derived requirements
- Drawn white box sequences
- Created ports and interfaces

The next step is to define – where appropriate – subsystem-level use cases.

It would be quite unusual for two different approaches to be both taken to allocating features to subsystems. In this Deskbook, we are doing so for pedagogical reasons. The **Start Up** use case is being done bottom-up and the (yet to come) **Control Air Surfaces** allocation will be done top-down. Usually, only one of these approaches would be taken for a given system.

Nevertheless, we'll try to show how this would work for the bottom-up approach and later for the top-down approach.

For the bottom-up approach, let's review what we've done so for in terms of allocations. First, we took our merged functional analysis and used the allocation wizard to allocate the features (attributes, operations, and events) to various subsystems. Then we created white box sequence diagrams to show how the allocated subsystems collaborate together to realize the system level use case scenarios.

At this point, we've allocated quite a number of elements to the subsystems. For example, the Figure 192 shows the model features allocated to the **ACES_Management** subsystem.

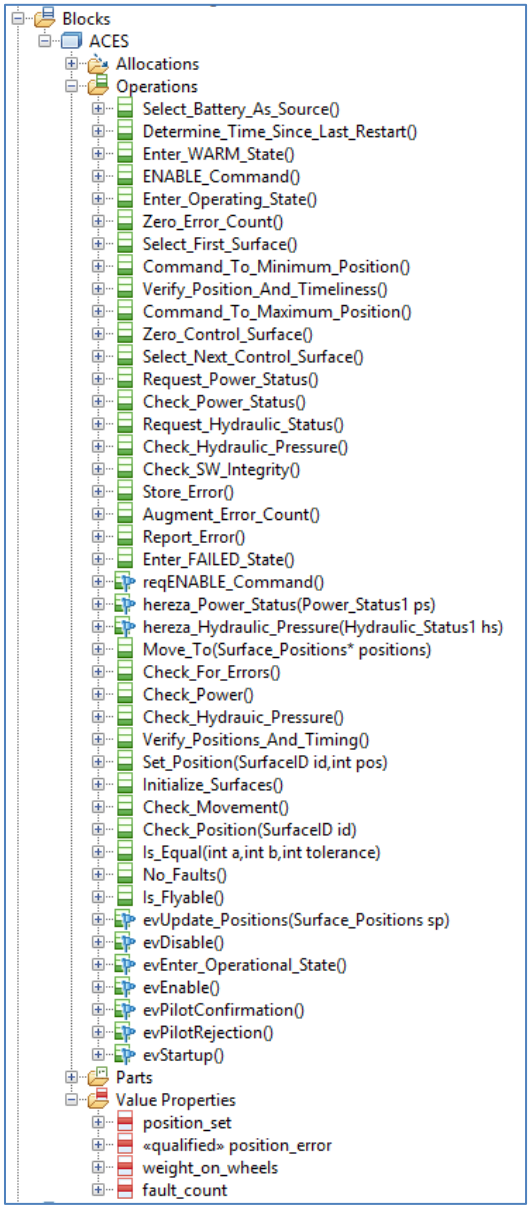


Figure 192: Model features allocated to the ACES_Management subsystem

Note that we haven't done the white box allocation for the **Control Air Surfaces** use case yet, nor added features from the other (as yet unanalyzed) use cases. You can see that it is already getting a bit complex. Grouping these features up into use cases makes sense from an organization point of view. On the other hand, very little is allocated to the hydraulic system, so we may not even need to create subsystem level use cases for that subsystem.

It's time to create some subsystem-level use cases to organize the requirements and features allocated to the **ACES_Management** subsystem.

- In the browser right-click on the **DesignSynthesisPkg > ArchitecturalDesignPkg > ACESDeompositionPkg > ACES_ManagementPkg** and select *Add New > General Element > Package*.
- Name this package **UseCasePkg**.
- Right click on **UseCasePkg** and select *Add New > Diagrams > Use Case Diagram*.
- Name this diagram **ACES_Management Use Cases**.

Now we must think about what use cases the subsystem must fulfill in order to satisfy the allocated requirements (see Table 3) and features. Remember all features and requirements allocated to the subsystem must (if we're defining subsystem use cases here) be further allocated to its use cases.

Now create the use case diagram for the subsystem as shown in Figure 193. A few things to note about the diagram.

- The **AMS** actor is the original actor located in the project-level **ActorPkg** package.
- The other actors are the peer subsystems
  - **aMS_Power**[18] represents the **ACES_Power** subsystem
  - **aMS_Hydraulics** represents the **ACES_Hydraulics** subsystem
  - **aMS_Control_Surface** represents **ACES_ControlSurface** subsystem
- These actors are stereotyped as **«internal»** to clearly differentiate them from the system actors[19].
- The use cases are stereotyped as **«Subsystem»** to indicate their scope of concern.

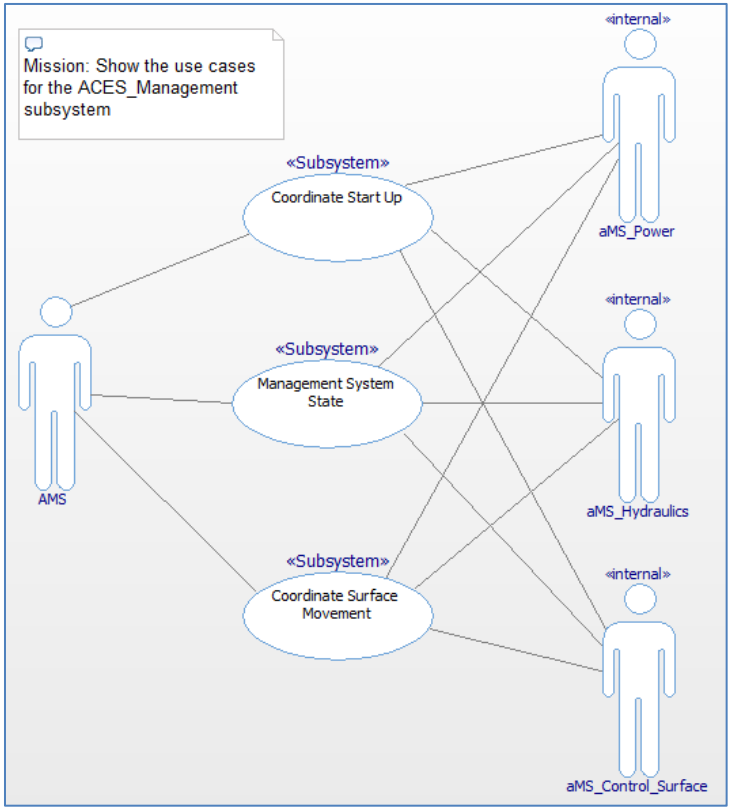


Figure 193: ACES_Management Subsystem use cases

9.3.1.5 Allocation Requirements to the Subsystem Use Cases

The next step is to allocate the requirements allocated to the subsystem to the use cases own by that subsystem. The easiest way to do that is to create

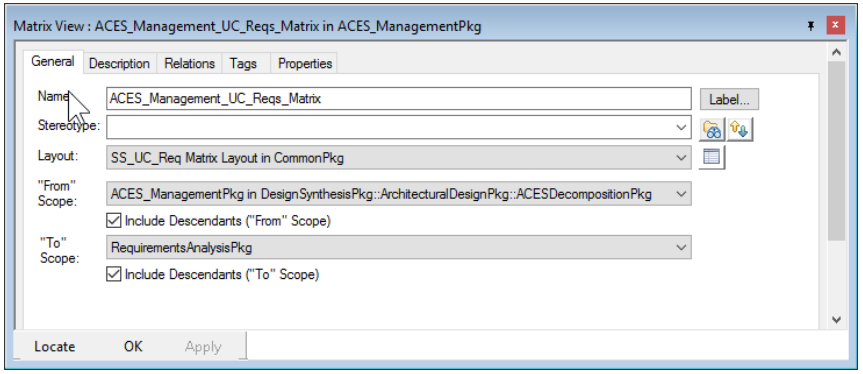
[18] The name of the subsystem is prefaces with "a" to indicate that it is being considered as an actor in this context.

[19] This is a stereotype I added to the **CommonPkg** package and it applies only to Actors.

a matrix that shows all the requirements allocated to the subsystem but also shows the use cases. Then walk through those requirements, one by one, and decide which subsystem use case it should be allocated to.

Such as matrix is a simple extension of the subsystem-requirements allocation matrix. The first difference in the layout definition is that the From element includes *Block*, *Class* and *Use Case*, rather than just *Block* and *Class*. The second difference is that in the *Cell Element Types* both *Allocation* and *Trace* are selected. The requirements are allocated to the susbsystem but will be traced to the use case, although note that in both cases the relation comes from the block or use case and ends on the requirement. This matrix layout definition should be placed in the **CommonPkg**.

The view, placed in the package created to hold the **ACES_Management** subsystem specification, uses the subsystem package as the *From* scope and the **RequirementsAnalysisPkg** and the *To* scope.



The matrix view showing the allocation to the **ACES_Management** subsystem and the trace to its use cases is shown below. Note the different icons for the allocation and trace relation in the matrix. Also be aware that the rows and columns are switched for the matrix.

	ACES_Management	Coordinate Start Up	Management System State	Coordinate Surface Movement
AM_requirement_1	AM_requirement_1			AM_requirement_1
ACSCUNT_requirement_3	ACSCUNT_requirement_3			ACSCUNT_requirement_3
AM_requirement_4	AM_requirement_4			AM_requirement_4
AM_requirement_6	AM_requirement_6			AM_requirement_6
ACSCUNT_requirement_7				
AM_requirement_9	AM_requirement_9			AM_requirement_9
ACSCUNT_requirement_10				
ACSCUNT_requirement_11				
ACSCUNT_requirement_12	ACSCUNT_requirement_12			ACSCUNT_requirement_12
ACSCUNT_requirement_13	ACSCUNT_requirement_13			ACSCUNT_requirement_13
ACSCUNT_requirement_16	ACSCUNT_requirement_16			ACSCUNT_requirement_16
ACSCUNT_requirement_17	ACSCUNT_requirement_17			ACSCUNT_requirement_17
ACSCUNT_requirement_18	ACSCUNT_requirement_18			ACSCUNT_requirement_18
ACSCUNT_requirement_19	ACSCUNT_requirement_19			
ACSCUNT_requirement_20	ACSCUNT_requirement_20		ACSCUNT_requirement_20	
ACSCUNT_requirement_21	ACSCUNT_requirement_21		ACSCUNT_requirement_21	
ACSCUNT_requirement_24	ACSCUNT_requirement_24			ACSCUNT_requirement_24
ACSCUNT_requirement_25				
ACSCUNT_requirement_26	ACSCUNT_requirement_26			ACSCUNT_requirement_26
AM_requirement_27	AM_requirement_27			AM_requirement_27
AM_requirement_28	AM_requirement_28		AM_requirement_28	
AM_requirement_29	AM_requirement_29		AM_requirement_29	
AM_requirement_30	AM_requirement_30		AM_requirement_30	
ACES_SS_requirement_32	ACES_SS_requirement_32	ACES_SS_requirement_32	ACES_SS_requirement_32	
ACES_SS_requirement_33	ACES_SS_requirement_33	ACES_SS_requirement_33		
ACES_SS_requirement_34	ACES_SS_requirement_34			
AM_requirement_35	AM_requirement_35	AM_requirement_35	AM_requirement_35	
DerIntReq_1				
DerIntReq_2				
DerIntReq_3				
DerIntReq_4				
DerIntReq_5				
DerIntReq_6				
DerIntReq_7	DerIntReq_7			DerIntReq_7
DerIntReq_8				
DerIntReq_9	DerIntReq_9		DerIntReq_9	
DerIntReq_10			DerIntReq_10	
DerIntReq_11	DerIntReq_11		DerIntReq_11	
DerIntReq_12	DerIntReq_12		DerIntReq_12	
DerReqInt_13	DerReqInt_13		DerReqInt_13	
DerIntReq_14	DerIntReq_14		DerIntReq_14	
DerIntReq_15			DerIntReq_15	
DerIntReq_16	DerIntReq_16		DerIntReq_16	
DerIntReq_17	DerIntReq_17		DerIntReq_17	
DerIntReq_18	DerIntReq_18		DerIntReq_18	
DerFunReq_1				
DerConfigReq_1				
DerConfigReq_2				
DerStartUpReq_1				
DerStartupReq_2				
DerStartupReq_3				
DerStartupReq_4	DerStartupReq_4		DerStartupReq_4	
InterfaceReq_0	InterfaceReq_0			InterfaceReq_0
InterfaceReq_1	InterfaceReq_1			InterfaceReq_1
InterfaceReq_2	InterfaceReq_2		InterfaceReq_2	
InterfaceReq_3	InterfaceReq_3		InterfaceReq_3	
InterfaceReq_4	InterfaceReq_4		InterfaceReq_4	
InterfaceReq_5	InterfaceReq_5		InterfaceReq_5	
InterfaceReq_6	InterfaceReq_6		InterfaceReq_6	
InterfaceReq_7	InterfaceReq_7		InterfaceReq_7	
InterfaceReq_8	InterfaceReq_8		InterfaceReq_8	

	ACES_Management	Coordinate Start Up	Management System State	Coordinate Surface Movement
InterfaceReq_9	InterfaceReq_9		InterfaceReq_9	
InterfaceReq_10	InterfaceReq_10		InterfaceReq_10	
InterfaceReq_11	InterfaceReq_11		InterfaceReq_11	
InterfaceReq_12	InterfaceReq_12		InterfaceReq_12	
InterfaceReq_13	InterfaceReq_13		InterfaceReq_13	
InterfaceReq_14	InterfaceReq_14		InterfaceReq_14	
InterfaceReq_15	InterfaceReq_15		InterfaceReq_15	
InterfaceReq_16	InterfaceReq_16		InterfaceReq_16	
InterfaceReq_17	InterfaceReq_17		InterfaceReq_17	
InterfaceReq_18	InterfaceReq_18		InterfaceReq_18	
InterfaceReq_19	InterfaceReq_19		InterfaceReq_19	
InterfaceReq_20	InterfaceReq_20		InterfaceReq_20	
InterfaceReq_21	InterfaceReq_21		InterfaceReq_21	
InterfaceReq_22	InterfaceReq_22		InterfaceReq_22	
InterfaceReq_23	InterfaceReq_23		InterfaceReq_23	
InterfaceReq_24	InterfaceReq_24		InterfaceReq_24	
InterfaceReq_25	InterfaceReq_25		InterfaceReq_25	
InterfaceReq_26	InterfaceReq_26		InterfaceReq_26	
InterfaceReq_27	InterfaceReq_27			
InterfaceReq_28	InterfaceReq_28			
InterfaceReq_29	InterfaceReq_29		InterfaceReq_29	
FuncReq_0	FuncReq_0		FuncReq_0	
FuncReq_1	FuncReq_1		FuncReq_1	
FuncReq_2	FuncReq_2		FuncReq_2	
FuncReq_3	FuncReq_3		FuncReq_3	
FuncReq_4	FuncReq_4		FuncReq_4	
FuncReq_5	FuncReq_5		FuncReq_5	
FuncReq_6	FuncReq_6		FuncReq_6	
FuncReq_7	FuncReq_7		FuncReq_7	
FuncReq_8	FuncReq_8		FuncReq_8	
FuncReq_9	FuncReq_9		FuncReq_9	
FuncReq_10	FuncReq_10		FuncReq_10	
FuncReq_11	FuncReq_11		FuncReq_11	
FuncReq_12	FuncReq_12		FuncReq_12	
FuncReq_13	FuncReq_13		FuncReq_13	
FuncReq_15	FuncReq_15		FuncReq_15	
FuncReq_16	FuncReq_16		FuncReq_16	
FuncReq_17	FuncReq_17		FuncReq_17	
FuncReq_18	FuncReq_18		FuncReq_18	
FuncReq_19	FuncReq_19		FuncReq_19	
FuncReq_20	FuncReq_20		FuncReq_20	
FuncReq_21	FuncReq_21		FuncReq_21	
FuncReq_22	FuncReq_22		FuncReq_22	
FuncReq_23	FuncReq_23		FuncReq_23	
FuncReq_24	FuncReq_24		FuncReq_24	
FuncReq_25	FuncReq_25		FuncReq_25	
FuncReq_26	FuncReq_26		FuncReq_26	
FuncReq_27	FuncReq_27		FuncReq_27	
FuncReq_28	FuncReq_28		FuncReq_28	
FuncReq_29	FuncReq_29		FuncReq_29	
FuncReq_30	FuncReq_30		FuncReq_30	
FuncReq_31	FuncReq_31		FuncReq_31	
FuncReq_32	FuncReq_32		FuncReq_32	
FuncReq_33	FuncReq_33		FuncReq_33	
FuncReq_34	FuncReq_34		FuncReq_34	
FuncReq_35	FuncReq_35		FuncReq_35	
FuncReq_36	FuncReq_36		FuncReq_36	
FuncReq_37	FuncReq_37		FuncReq_37	
FuncReq_38	FuncReq_38		FuncReq_38	
FuncReq_39	FuncReq_39		FuncReq_39	

	ACES_Management	Coordinate Start Up	Management System State	Coordinate Surface Movement
FuncReq_40	FuncReq_40		FuncReq_40	
FuncReq_100				
FuncReq101				
FuncReq201				
FuncReq202				
FuncReq203				
ShutDownReq_0				
ShutDownReq_1				
ShutDownReq_2				
MaintenanceReq_0				
MaintenanceReq_1				
MaintenanceReq_2				
StateModesReq_0				StateModesReq_0
StateModesReq_1				StateModesReq_1
StateModesReq_2				StateModesReq_2
StateModesReq_3				StateModesReq_3
StateModesReq_4				StateModesReq_4
StateModesReq_5				StateModesReq_5
StateModesReq_6				
ErrorReq_0	ErrorReq_0			ErrorReq_0
ErrorReq_1	ErrorReq_1			ErrorReq_1
ErrorReq_2	ErrorReq_2			ErrorReq_2
ErrorReq_3	ErrorReq_3			ErrorReq_3
ErrorReq_4	ErrorReq_4			ErrorReq_4
ErrorReq_5	ErrorReq_5			ErrorReq_5
ErrorReq_6	ErrorReq_6			ErrorReq_6
ErrorReq_7	ErrorReq_7			ErrorReq_7
ErrorReq_8	ErrorReq_8			ErrorReq_8
ErrorReq_9	ErrorReq_9			ErrorReq_9
ErrorReq_10	ErrorReq_10			ErrorReq_10
ErrorReq_11	ErrorReq_11			ErrorReq_11
ErrorReq_12	ErrorReq_12			ErrorReq_12
ErrorReq_13	ErrorReq_13			ErrorReq_13
ErrorReq_14	ErrorReq_14			ErrorReq_14
ErrorReq_15	ErrorReq_15			ErrorReq_15
ErrorReq_16	ErrorReq_16			ErrorReq_16
ErrorReq_17	ErrorReq_17			ErrorReq_17
ErrorReq_18	ErrorReq_18			ErrorReq_18
ErrorReq_19	ErrorReq_19			ErrorReq_19
ErrorReq_20	ErrorReq_20			ErrorReq_20
ErrorReq_21	ErrorReq_21			ErrorReq_21
ErrorReq_22	ErrorReq_22			ErrorReq_22
ErrorReq_23	ErrorReq_23			ErrorReq_23
ErrorReq_24	ErrorReq_24			ErrorReq_24
ErrorReq_25	ErrorReq_25			ErrorReq_25
ErrorReq_26	ErrorReq_26			ErrorReq_26
ErrorReq_27	ErrorReq_27			ErrorReq_27
ErrorReq_28	ErrorReq_28			ErrorReq_28
ErrorReq_29	ErrorReq_29			ErrorReq_29
ErrorReq_30	ErrorReq_30			ErrorReq_30
ErrorReq_31	ErrorReq_31			ErrorReq_31
ErrorReq_32	ErrorReq_32			ErrorReq_32
ErrorReq_33	ErrorReq_33			ErrorReq_33
ErrorReq_34	ErrorReq_34			ErrorReq_34
ErrorReq_35	ErrorReq_35			ErrorReq_35
ErrorReq_36	ErrorReq_36			ErrorReq_36
ErrorReq_37	ErrorReq_37			ErrorReq_37
ConfigReq_0				
ConfigReq_1	ConfigReq_1			ConfigReq_1
ConfigReq_2				ConfigReq_2
ConfigReq_3	ConfigReq_3			ConfigReq_3

	ACES_Management	Coordinate Start Up	Management System State	Coordinate Surface Movement
OtherReq_0	OtherReq_0	OtherReq_0	OtherReq_0	OtherReq_0
OtherReq_1	OtherReq_1	OtherReq_1	OtherReq_1	OtherReq_1
StartUpReq_0				
StartUpReq_1	StartUpReq_1			StartUpReq_1
StartUpReq_2	StartUpReq_2			StartUpReq_2
StartUpReq_3	StartUpReq_3			StartUpReq_3
StartUpReq_4	StartUpReq_4			StartUpReq_4
StartUpReq_5	StartUpReq_5			StartUpReq_5
StartUpReq_6	StartUpReq_6			StartUpReq_6
SafetyReq_390197	SafetyReq_390197			SafetyReq_390197
Safety_Req_390198	Safety_Req_390198			Safety_Req_390198
Safety_Req_390199	Safety_Req_390199			Safety_Req_390199
Safety_Req_390200	Safety_Req_390200			Safety_Req_390200
Safety_Req_390201	Safety_Req_390201			Safety_Req_390201
Safety_Req_390202				
Safety_Req_390203				
Safety_Req_390204				
Safety_Req_390205				
Safety_Req_390206	Safety_Req_390206		Safety_Req_390206	
Safety_Req_390207	Safety_Req_390207			Safety_Req_390207
Safety_Req_390208	Safety_Req_390208			Safety_Req_390208
Safety_Req_390209	Safety_Req_390209			Safety_Req_390209
Safety_Req_390210	Safety_Req_390210			Safety_Req_390210
Safety_Req_390211				
Safety_Req_390212	Safety_Req_390212		Safety_Req_390212	
Safety_Req_390213	Safety_Req_390213		Safety_Req_390213	
Safety_Req_390214	Safety_Req_390214			Safety_Req_390214
Safety_Req_390215	Safety_Req_390215			Safety_Req_390215
Safety_Req_390216	Safety_Req_390216			Safety_Req_390216
Safety_Req_390217				
Safety_Req_390218				
SafetyReq_001	SafetyReq_001			SafetyReq_001
SafetyReq_002	SafetyReq_002			SafetyReq_002
SafetyReq_003	SafetyReq_003			SafetyReq_003
SafetyReq_004	SafetyReq_004			SafetyReq_004
SafetyReq_005	SafetyReq_005			SafetyReq_005
SafetyReq_006				

Figure 194: ACES_Management Use Cases Requirements Trace

9.3.2 Top-Down Approach: Control Air Surfaces Use Case

The top-down approach works by directly creating the subsystem-level use cases directly from the system level use case. The «include» relation is used for this. The subsystem level use cases represent coherent sets of requirements and behaviors that apply to a single subsystem. Once identified, the subsystem use cases are allocated to the subsystems and they are then elaborated in the same fashion as the system use cases.

We recommend that there is one (or possibly more) diagram for each system level use case's decomposition to subsystem use cases. If the diagram is not too complex, then the allocations to the subsystems may be included on this diagram as well. Otherwise, simply create another view (diagram or table) for the allocation. These diagrams and use cases need some place to live so in the **DesignSynthesisPkg > ArchitecturalDesignPkg** create a new package named **SubsystemUseCasesPkg**. The use case diagrams that show the relations of the system and subsystem use cases will live here but later the subsystem use cases themselves will be moved into the packages that specify their owning subsystems. This package will also hold the internal actors (representing the peer subsystems.

9.3.2.1 Decompose Use Cases

The **Control Air Surfaces** use case decomposition is shown in Figure 195. There are a few noteworthy aspects to the diagram. First, the actors include both the system actors (**AMS** and **Pilot_Display**) but also the internal actors, which are stand-ins for the peer subsystems.

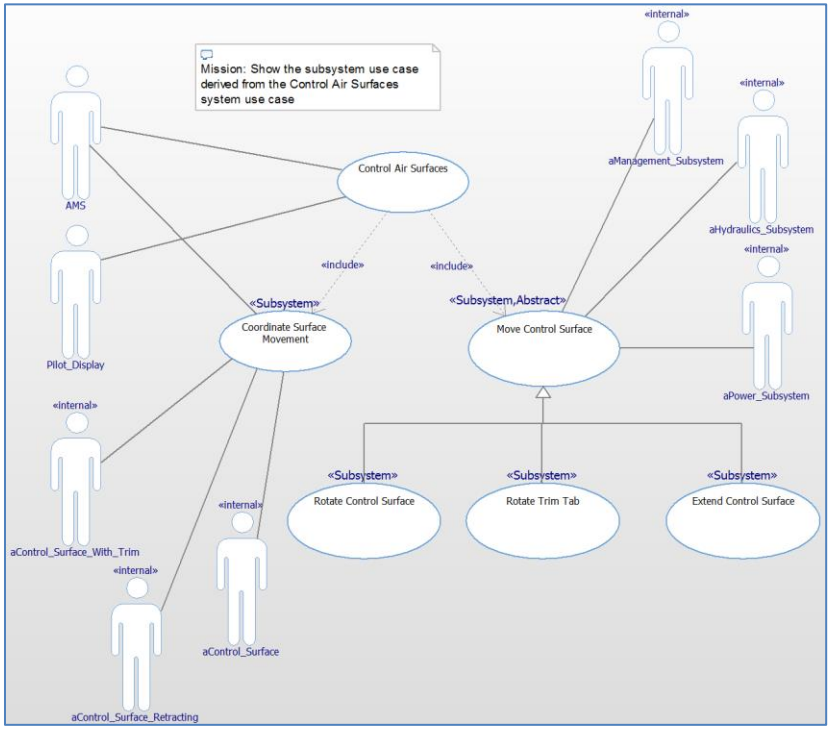


Figure 195: Decomposition of the Control Air Surface Use Case

Next, note that there are two directly included subsystem level use cases derived from the **Control Air Surfaces** use case. The first is **Coordinate Surface Movement**. The other is **Move Control Surface**. This latter use case is stereotyped both **«Subsystem»** and **«Abstract»**. This latter stereotype is meant to indicate that the use case is really just a place holder and contains no requirements or specification itself; rather it is there only to help organize the use cases derived from it[20].

These use cases, **Rotate Control Surface**, **Rotate Trim Tab**, and **Extend Control Surface** are specializations of **Move Control Surface**. Why use this intermediary use case when it doesn't contain any actual requirements? The reason is that each of these specialized use cases associates with three internal actors: **aManagementSubsystem**, **aHydraulics**, and **aPower**. Because of the way specialization works, each of the specialized subsystem use cases inherits the relations to these actors, so that we don't have to draw these relations to each of the more specialize use cases. Esssentially, it is being used as a "notational convenience."

Each of these subsystem use cases is allocatable to a single subsystem. This allocation is shown in the next diagram, Figure 196.

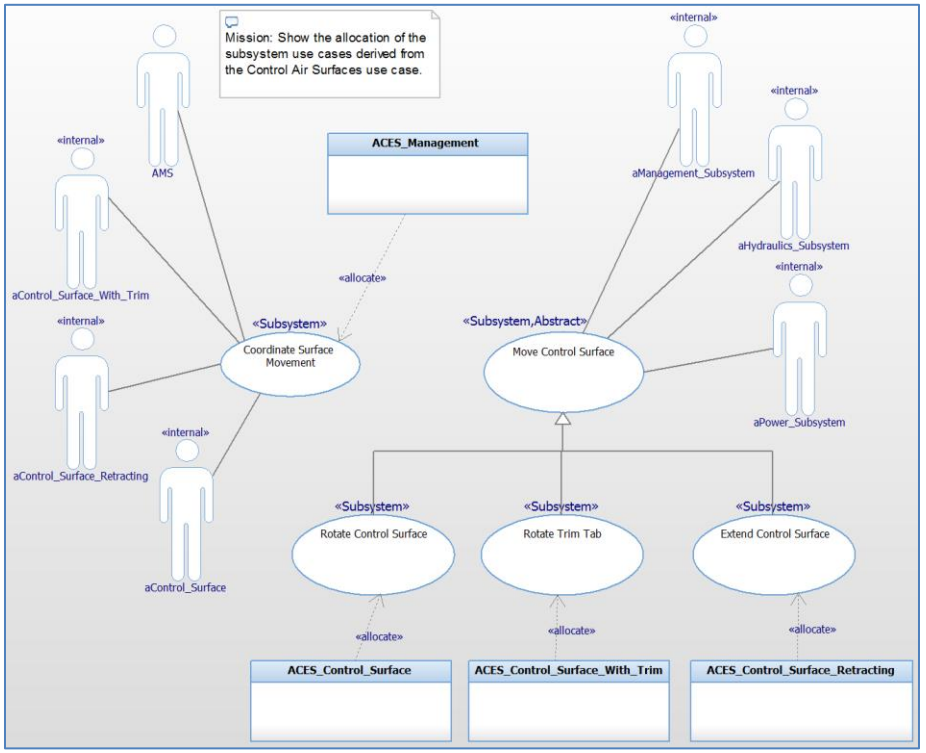


Figure 196: Allocation of subsystem use cases

Once these use cases are allocated to subsystems, they should be moved to the packages holding the subsystem blocks. This allows each of these packages to hold the specification of that subsystem. This organization will be important later when we hand off these specifications to the subsystem teams for further design and implementation. Be aware that the **Move Control Surfaces** use case, being abstract, is not allocated and so can remain in the **SubsystemUseCasesPkg**.

Now, for each subsystem, we draw one (or more) use case diagram to show the set of use cases allocated to the subsystem, similar to Figure 193. Since we've only done a little of the work, this diagram for the **ACES_ControlSurface** will be a bit sparse. As we repeat this procedure with other system level use cases, we'll add other subsystem level use cases here, such as **Configure Movement**, **Perform Self Test**, and so on which we

[20] This stereptype applies to use cases only and is also put into the **CommonPkg**.

anticipate will be detailed later. This is shown in Figure 197. Some yet-to-be identified use cases are shown, just to give you a hint of what might be identified in further iterations. At this point, these other use case are notional. Also note that the relations between **Rotate Control Surface** use case and the actors are inherited because it is a specialization of **Move Control Surface** use case.

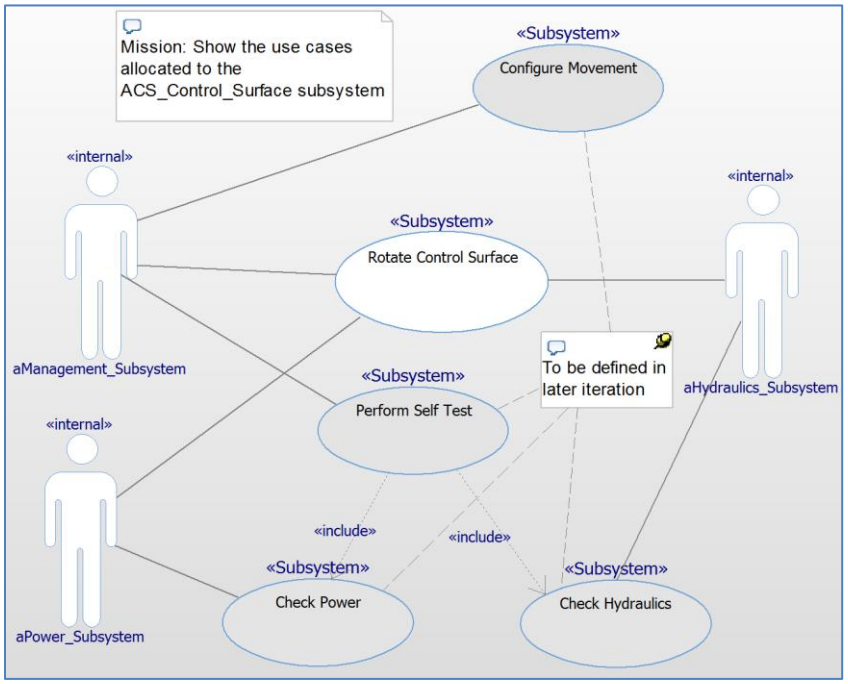
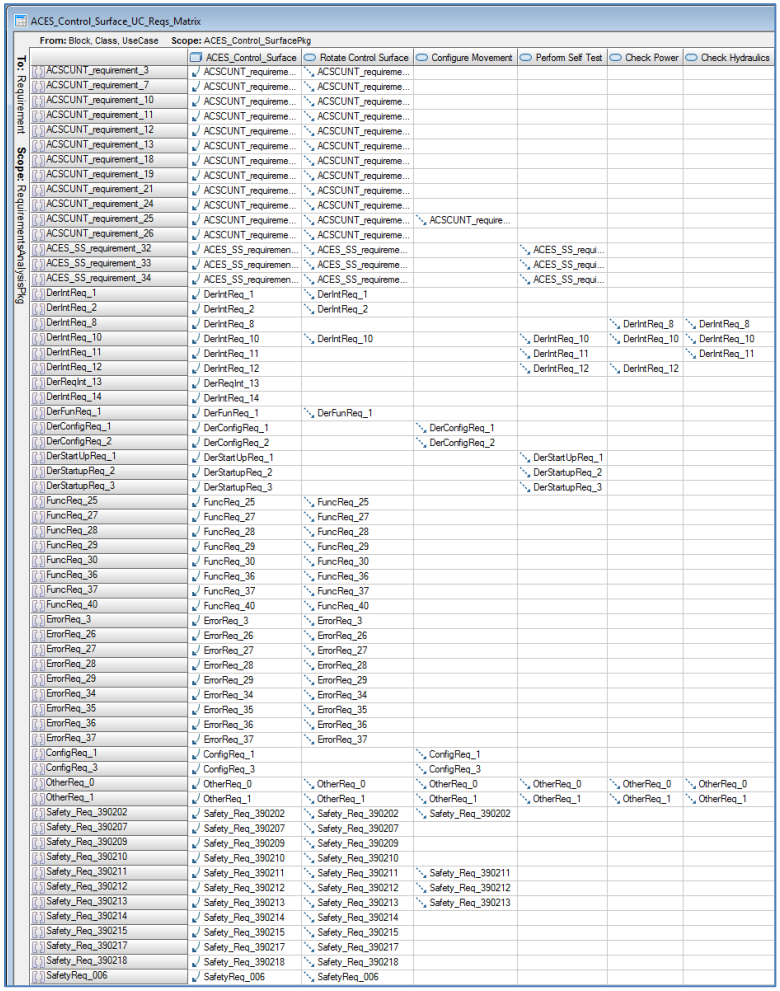


Figure 197: ACES_ControlSurface Subsystem Use Cases

9.3.2.2 Allocate Requirements to the Subsystem Use Cases

Using the same matrix layout as in Section 9.3.1.5, we create a matrix view of the requirements relevant to the **ACES_ControlSurface** subsystem. The matrix view below shows the requirements allocated to the **ACES_ControlSurface** subsystem and traced to its use cases (with *Toggle Empty Rows* toggled off). Note the different iconic symbol in the cells identifying the different relations (*allocation* to the subsystem and *trace* to the use case).


ACES_Control_Surface_UC_Reqs_Matrix

From: Block, Class, UseCase Scope: ACES_Control_SurfacePkg

To: Requirement Scope: RequirementsAnalysisPkg

	ACES_Control_Surface	Rotate Control Surface	Configure Movement	Perform Self Test	Check Power	Check Hydraulics
ACSCUNT_requirement_3	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_7	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_10	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_11	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_12	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_13	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_18	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_19	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_21	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_24	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACSCUNT_requirement_25	ACSCUNT_requireme...	ACSCUNT_requireme...	ACSCUNT_require...			
ACSCUNT_requirement_26	ACSCUNT_requireme...	ACSCUNT_requireme...				
ACES_SS_requirement_32	ACES_SS_requiremen...	ACES_SS_requireme...		ACES_SS_requi...		
ACES_SS_requirement_33	ACES_SS_requiremen...	ACES_SS_requireme...		ACES_SS_requi...		
ACES_SS_requirement_34	ACES_SS_requiremen...	ACES_SS_requireme...		ACES_SS_requi...		
DerIntReq_1	DerIntReq_1	DerIntReq_1				
DerIntReq_2	DerIntReq_2	DerIntReq_2				
DerIntReq_8	DerIntReq_8				DerIntReq_8	DerIntReq_8
DerIntReq_10	DerIntReq_10	DerIntReq_10		DerIntReq_10	DerIntReq_10	DerIntReq_10
DerIntReq_11	DerIntReq_11			DerIntReq_11		DerIntReq_11
DerIntReq_12	DerIntReq_12			DerIntReq_12	DerIntReq_12	
DerReqInt_13	DerReqInt_13					
DerIntReq_14	DerIntReq_14					
DerFunReq_1	DerFunReq_1	DerFunReq_1				
DerConfigReq_1	DerConfigReq_1		DerConfigReq_1			
DerConfigReq_2	DerConfigReq_2		DerConfigReq_2			
DerStartUpReq_1	DerStartUpReq_1			DerStartUpReq_1		
DerStartupReq_2	DerStartupReq_2			DerStartupReq_2		
DerStartupReq_3	DerStartupReq_3			DerStartupReq_3		
FuncReq_25	FuncReq_25	FuncReq_25				
FuncReq_27	FuncReq_27	FuncReq_27				
FuncReq_28	FuncReq_28	FuncReq_28				
FuncReq_29	FuncReq_29	FuncReq_29				
FuncReq_30	FuncReq_30	FuncReq_30				
FuncReq_36	FuncReq_36	FuncReq_36				
FuncReq_37	FuncReq_37	FuncReq_37				
FuncReq_40	FuncReq_40	FuncReq_40				
ErrorReq_3	ErrorReq_3	ErrorReq_3				
ErrorReq_26	ErrorReq_26	ErrorReq_26				
ErrorReq_27	ErrorReq_27	ErrorReq_27				
ErrorReq_28	ErrorReq_28	ErrorReq_28				
ErrorReq_29	ErrorReq_29	ErrorReq_29				
ErrorReq_34	ErrorReq_34	ErrorReq_34				
ErrorReq_35	ErrorReq_35	ErrorReq_35				
ErrorReq_36	ErrorReq_36	ErrorReq_36				
ErrorReq_37	ErrorReq_37	ErrorReq_37				
ConfigReq_1	ConfigReq_1		ConfigReq_1			
ConfigReq_3	ConfigReq_3		ConfigReq_3			
OtherReq_0	OtherReq_0	OtherReq_0	OtherReq_0	OtherReq_0	OtherReq_0	OtherReq_0
OtherReq_1	OtherReq_1	OtherReq_1	OtherReq_1	OtherReq_1	OtherReq_1	OtherReq_1
Safety_Req_390202	Safety_Req_390202	Safety_Req_390202	Safety_Req_390202			
Safety_Req_390207	Safety_Req_390207	Safety_Req_390207				
Safety_Req_390209	Safety_Req_390209	Safety_Req_390209				
Safety_Req_390210	Safety_Req_390210	Safety_Req_390210				
Safety_Req_390211	Safety_Req_390211	Safety_Req_390211	Safety_Req_390211			
Safety_Req_390212	Safety_Req_390212	Safety_Req_390212	Safety_Req_390212			
Safety_Req_390213	Safety_Req_390213	Safety_Req_390213	Safety_Req_390213			
Safety_Req_390214	Safety_Req_390214	Safety_Req_390214				
Safety_Req_390215	Safety_Req_390215	Safety_Req_390215				
Safety_Req_390217	Safety_Req_390217	Safety_Req_390217				
Safety_Req_390218	Safety_Req_390218	Safety_Req_390218				
SafetyReq_006	SafetyReq_006	SafetyReq_006				

Figure 198: Requirements traces to ACES_ControlSurface use cases

You should also note that a number of control surface requirements – specifically those related to trim tabs and extension/retraction – are not represented here because they are allocated to the

ACES_ControlSurfaceWithTrim and **ACES_ControlSurfaceRetracting** subsystems, respectively.

9.3.2.3 Define Subsystem Use Case Analysis Context

Let's continue our focus on the **ACES_ControlSurfaces** subsystem use case **Rotate Control Surface**. Here we will define some scenarios for the use case.

Before we do that, we'll need to create a place to hold the functional analysis of the subsystem use cases that require it.

- In the **DesignSynthesisPkg > ArchitecturalDesignPkg > ACESDecompositionPkg > ACES_ControlSurfacePkg** create new nested package, named **FAPkg** (for **F**unctional **A**nalysis **P**ac**k**a**g**e).
- Inside of the new **FAPkg**, create a nested package for the analysis of this specific use case, named **RotateSurfacesPkg**. It is in this package that we will analysis this use case.

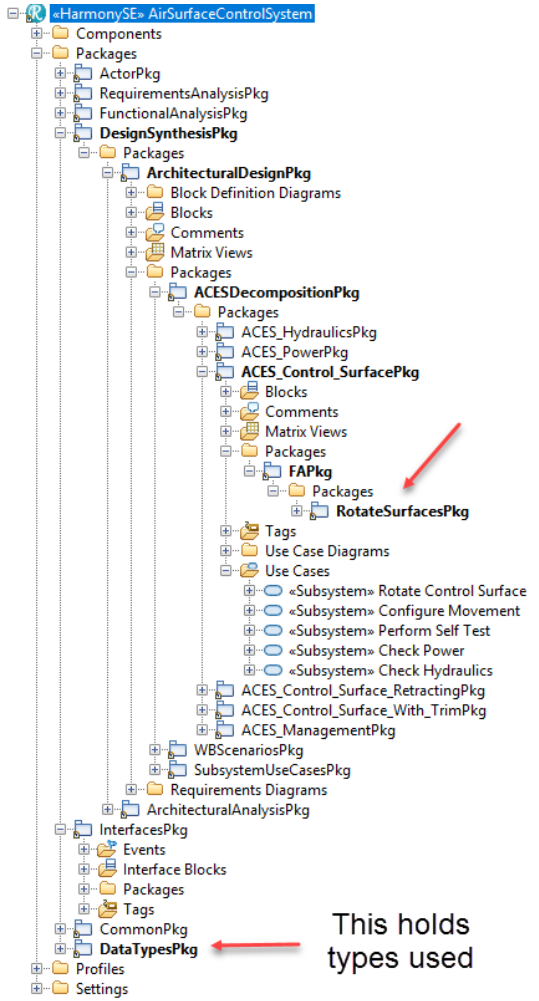


Figure 199: Package for analyzing the subsystem use case

Now we must add the use case context block diagram to this package, just as we did for the system use cases. Although in this case, we'll need to do it manually as the toolkit won't help us here. Be sure that when you specify the ports that you check the *Behavior* (all) and the *Conjugated* (actor blocks) check boxes on the proxy port features dialogs. Since the Ports and Interfaces wizard will create some of these ports later, you can defer adding the ports and connectors until after you've run the wizard, if you prefer.

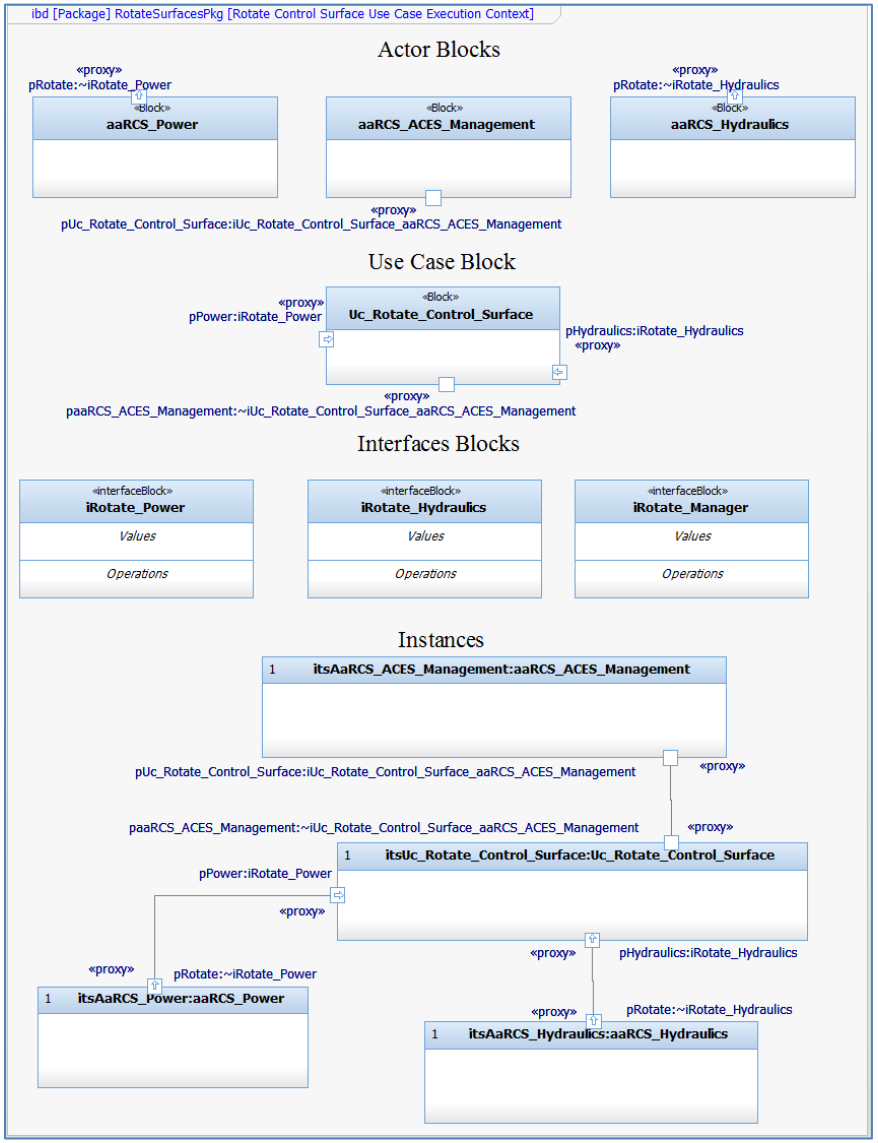
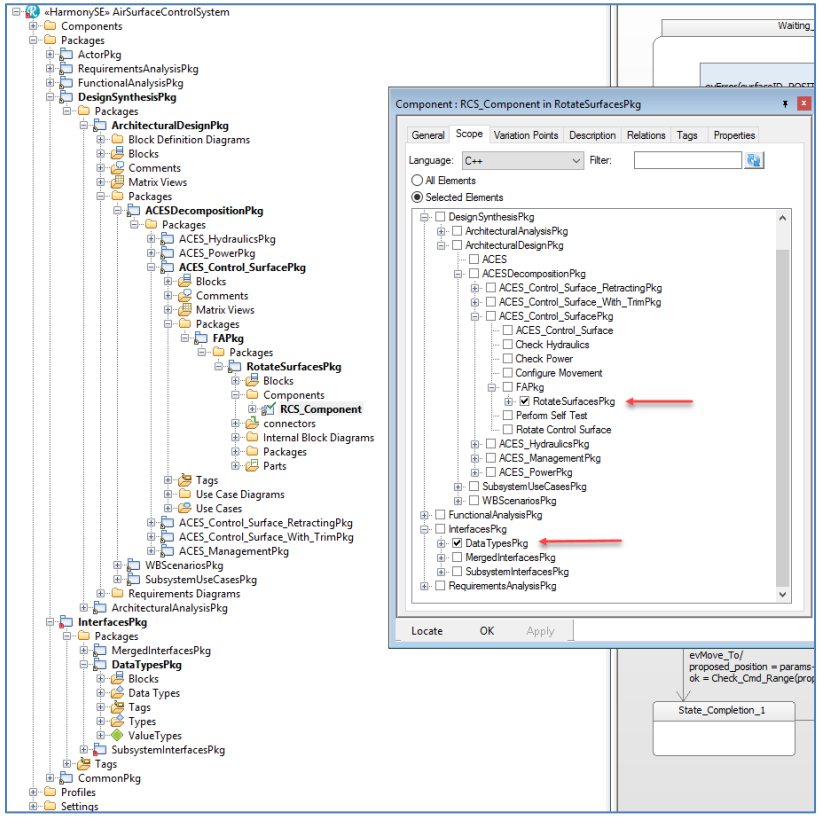


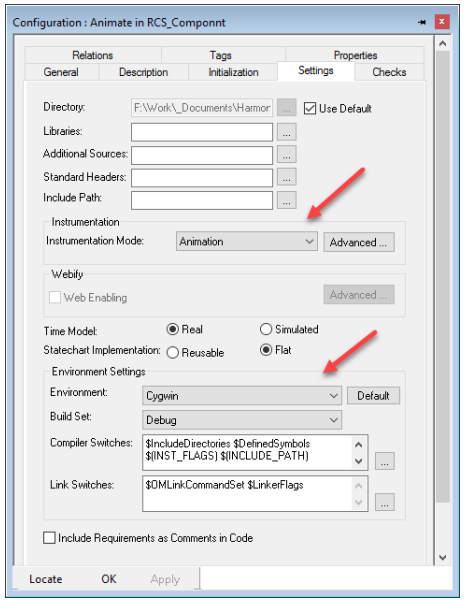
Figure 200: Rotate Control Surface Use Case Execution Context

We also need to add an execution component to support the simulation of this use case.

- In the **DesignSynthesisPkg > ArchitecturalDesignPkg > ACESDecompositionPkg > ACES_ControlSurfacePkg > FAPkg > RotateControlSurfacesPkg** package add a new component. Name this component **RCS_Component**
- In the **RCS_Component**, add the packages to support the execution, including the **InterfacesPkg > DataTypesPkg** package (we'll need this later)



- Rename the component configuration **Animate**.
- Set the *Settings* of the **Animate** configuration of the component to support animation (use your own selected compiler though):



The structure of the current structure of the **ACES_ControlSurface** subsystem packages looks like this:

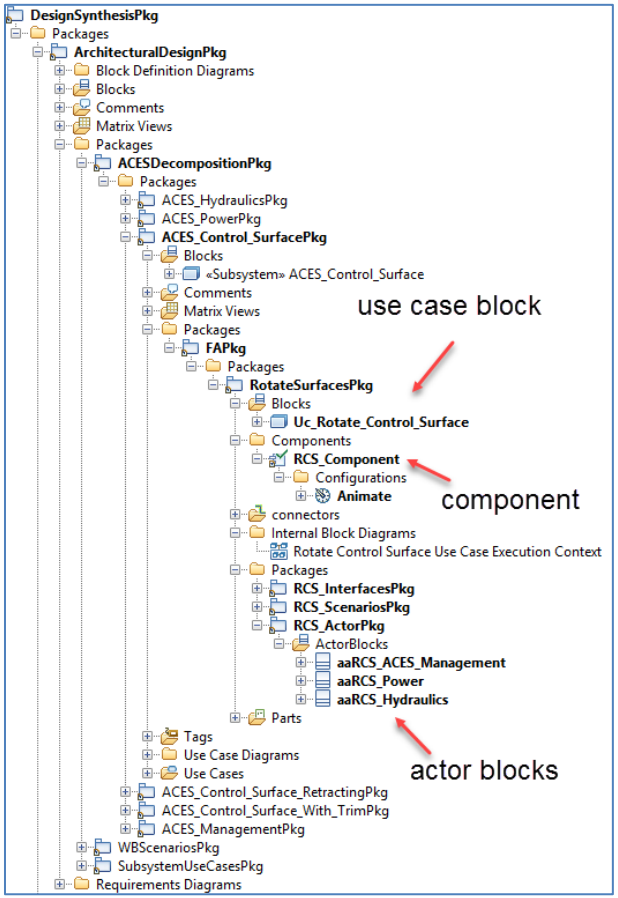


Figure 201: Current structure of the ACES_ControlSurface subsystem packages

We can use any of the alternative methods from Figure 4 but we'll continue to use the Interaction-based approach. That means the next step is to define scenarios.

9.3.2.4 Define Subsystem-Level Scenarios

In this set of scenarios, there are some sub-activities that go one in parallel, and we'll draw these as separate sequence diagrams references by the main flows.

The first is that there are requirements ACES system to report the surface positions on a periodic basis. This is passed down to the individual control surface subsystems to report their own positions on a periodic basis.

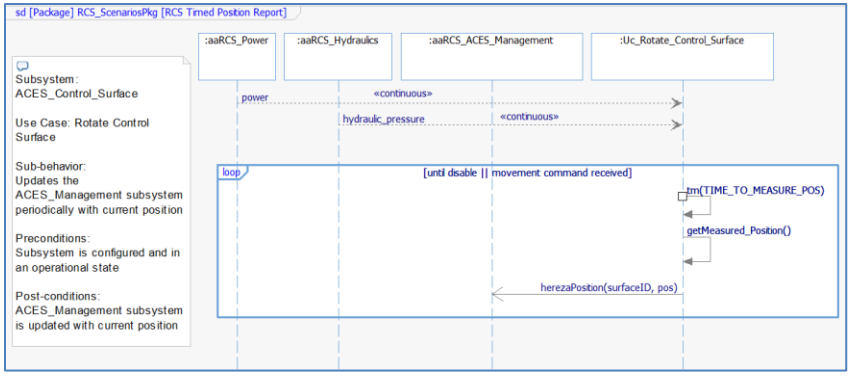


Figure 202: Interaction fragment for Timed Position Report

Next, when the subsystem is operational but not currently responding to a movement command, it must be performing *stationkeeping.* This means that the system must periodically make small adjustments to the surface position to keep that position correct even in the face of changing forces. Here's that interaction fragment.

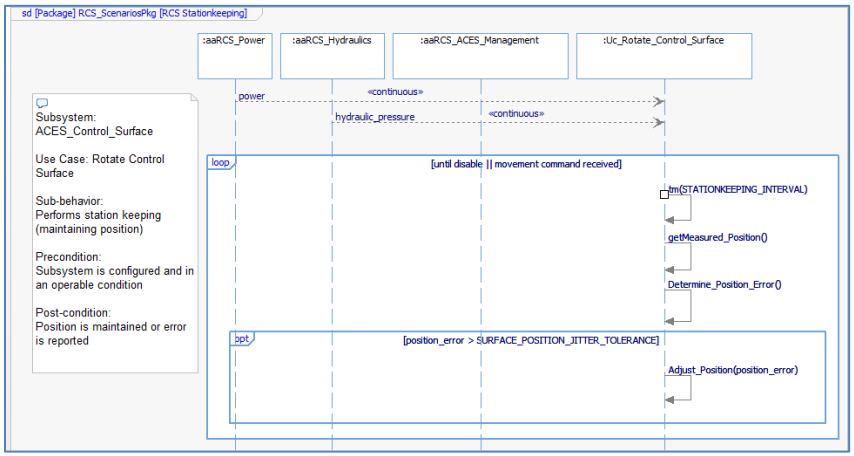


Figure 203: Interaction fragment for Stationkeeping

Now we can look at the main scenarios for this use case. The first, shows what happens when the subsystem receives a valid movement command:

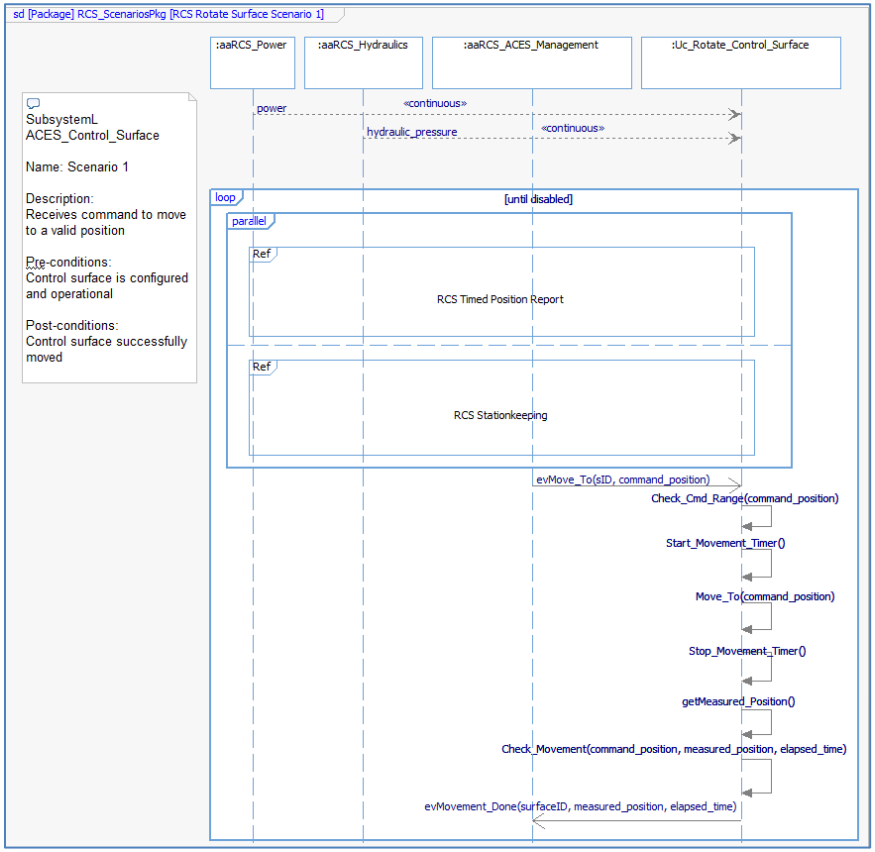


Figure 204: Rotate Control Surface Scenario 1

The second main scenario shows what happens when a command value is sent that is out of range:

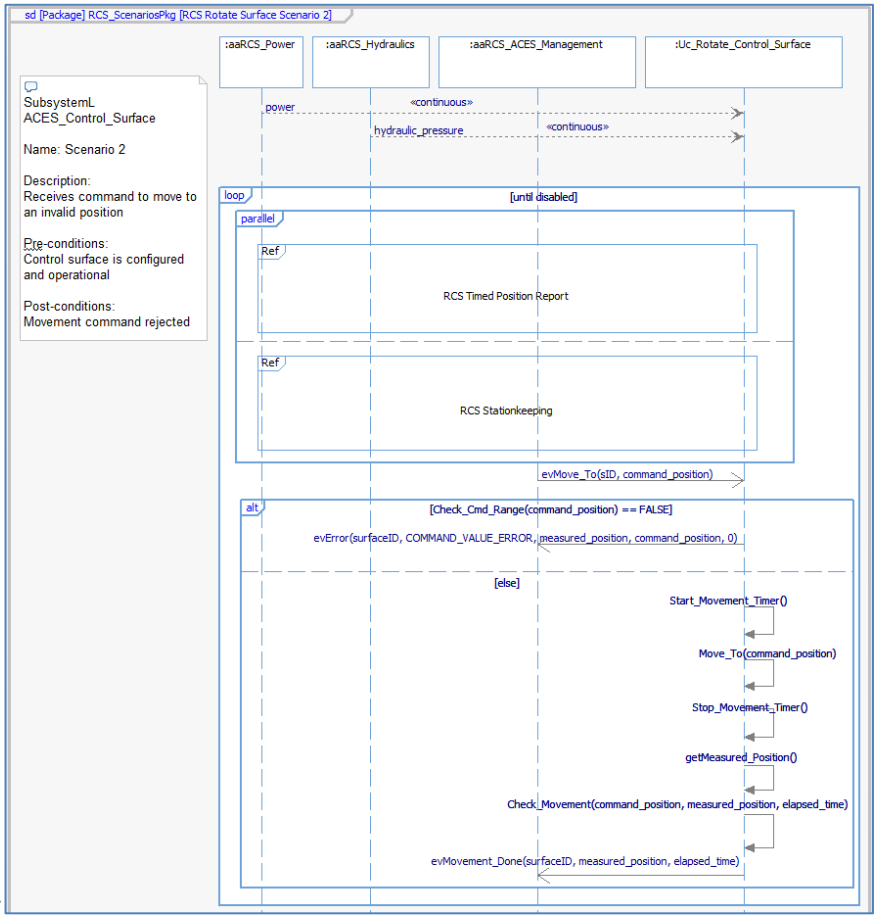


Figure 205: Rotate Control Surface Scenario 2

Note that the error passed in Scenario 2 comes from the **ERROR_TYPE** we defined in the **Start Up** use case package.

The last scenario in this set show what happens if either the commanded position is not achieved with enough accuracy or if the command position is achieved outside of the timing constaints:

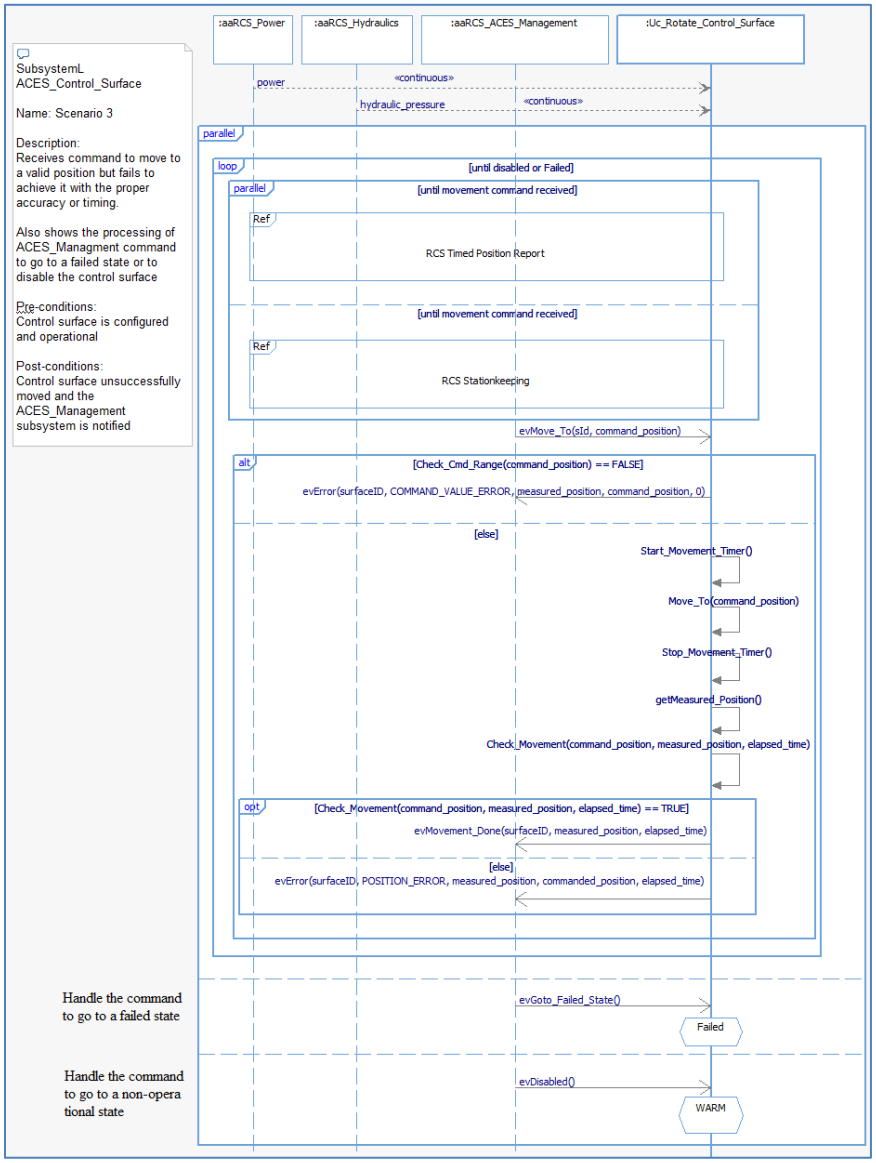


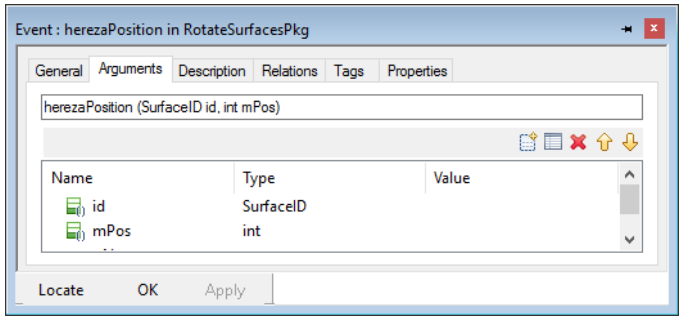
Figure 206: Rotate Control Surface Scenario 3

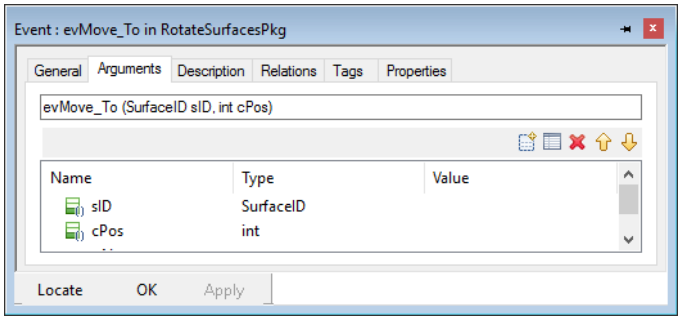
As before the error passed in Scenario 3 comes from the **ERROR_TYPE** previously defined for the **Start Up** use case.

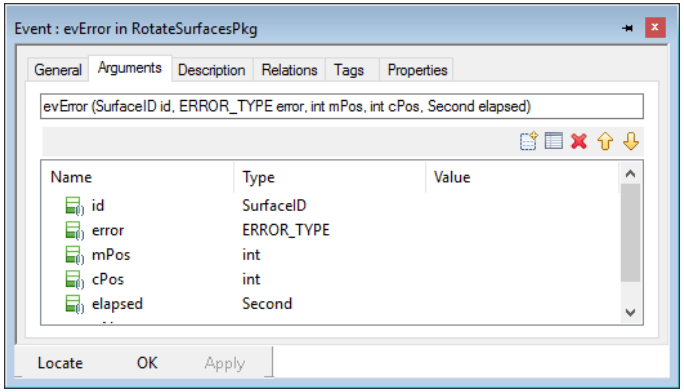
9.3.2.5 Define Subsystem Ports and Interfaces

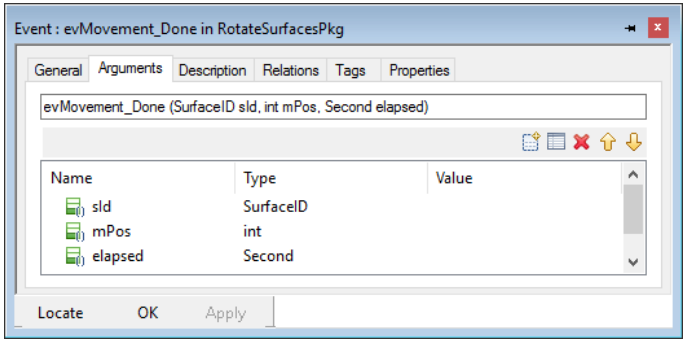
At this point the two different workflows join and proceed together. First, we'll use the *Create Ports and Interfaces* tool of the SE Toolkit as before. Since the interface blocks are defined in the **RCS_InterfacesPkg**, the toolkit will update these interface blocks with the events. Then we'll update the parameters of the events.

- Go to each of the sequence diagrams in the previous section, right click in the diagram and select *Auto Realize All Elements*. This will add the elements to the model from the sequence diagram.
- Right click the **DesignSynthesisPkg > ArchitecturalDesignPkg > ACESDecompositionPkg > ACES_ControlSurfacePkg > FAPkg > RotateControlSurfacesPkg > RCS_ScenariosPkg** and select *SE-Toolkit > Ports and Interfaces > Create Ports and Interfaces Recusive*
- Go to **DesignSynthesisPkg > ArchitecturalDesignPkg > ACESDecompositionPkg > ACES_ControlSurfacePkg > FAPkg > RotateControlSurfacesPkg > Events** and add the parameters to the events used in the sequence diagrams.
- Edit the event parameters as shown below. This may require using the *Select* option in the *Type* drop down list to navigate to the high-level **InterfacesPkg > DataTypesPkg** package.



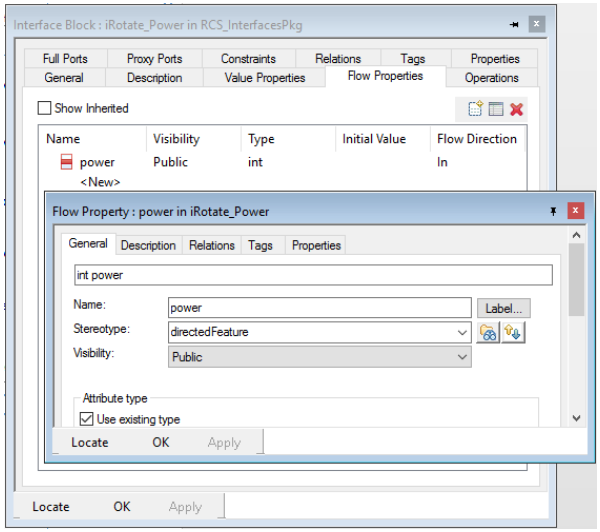


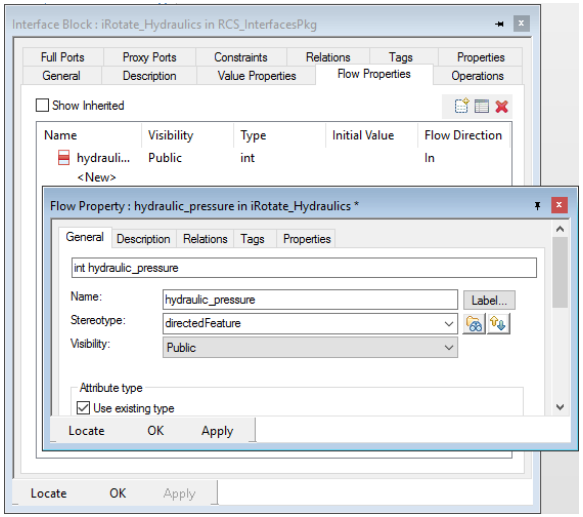




Last in this section, we must manually add the flows to the interface blocks, since the **AutoRealize All Elements** will not realize either of the **power** and **pressure** flows that begine each scenario. Note that they may (or may not) affect the execution, but they are still an important interface that should be specified.

- Right click on the **aaRCS_Hydraulics** actor block in the browser and select *Add New > Ports and Flows > Flow Property.* Name this property **hydraulic_pressure** (the default type of *int* is fine).
- Right click on the **aaRCS_Power** actor block in the browser and select *Add New > Ports and Flows > Flow Property.* Name this property **power** (the default type of *int* is fine).
- Add both **power** and **hydraulic_pressure** flow properties to the use case block **Uc_RotateControlSurface**.
- Add the pressure flow property to the interface block **iUc_RotateControlSurface_aaRCSHydraulics**. Set its direction to *in*.
- Add the pressure flow property to the interface block **iUc_RotateControlSurface_aaRCSHydraulics**. Set its direction to *in*.
- Set the stereotype of all of these flow properties to *«directedFeature»*





The features of the blocks and interface blocks should now look like Figure 207.

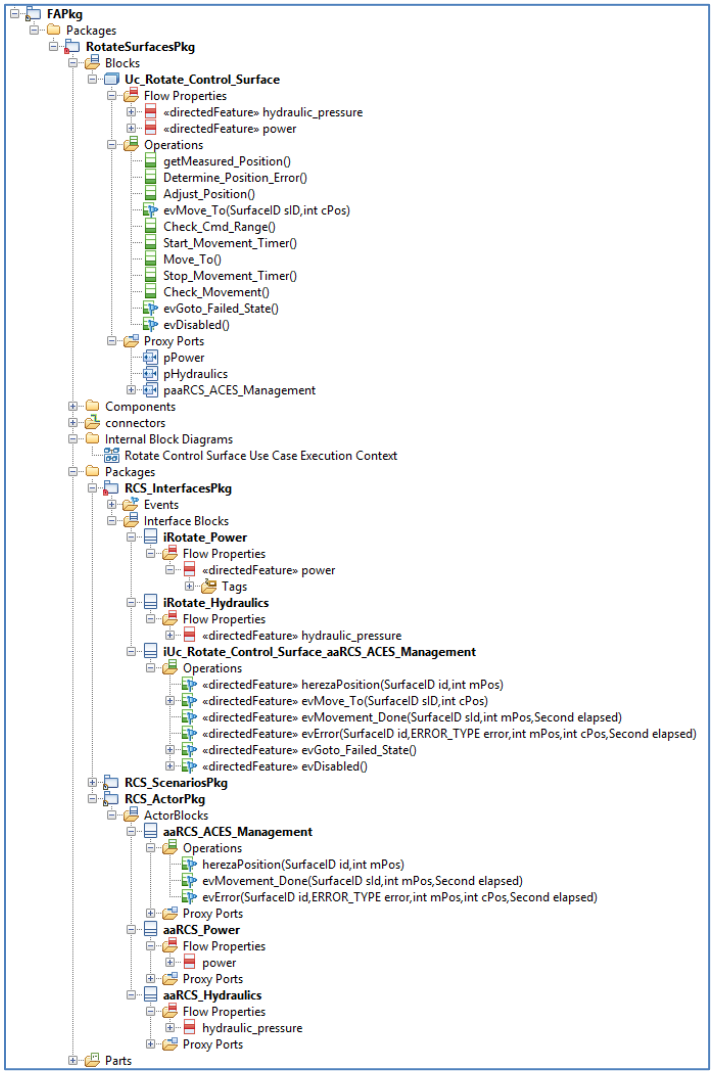


Figure 207: Block features for the Rotate Control Surface use case

9.3.3 Derive Subsystem Use Case State Behavior

We can now construct the state machine for this subsystem use cases. In this case, we'll build the **ACES_ControlSurface** subsystem use case **Rotate Control Surface** state machine, but we have a few others that we've identified that we could use as an example, such as the **ACES_Management** subsystem use case **Coordinate Surface Movement**.

The figure below shows the state machine for the **Rotate Control Surface** use case block:

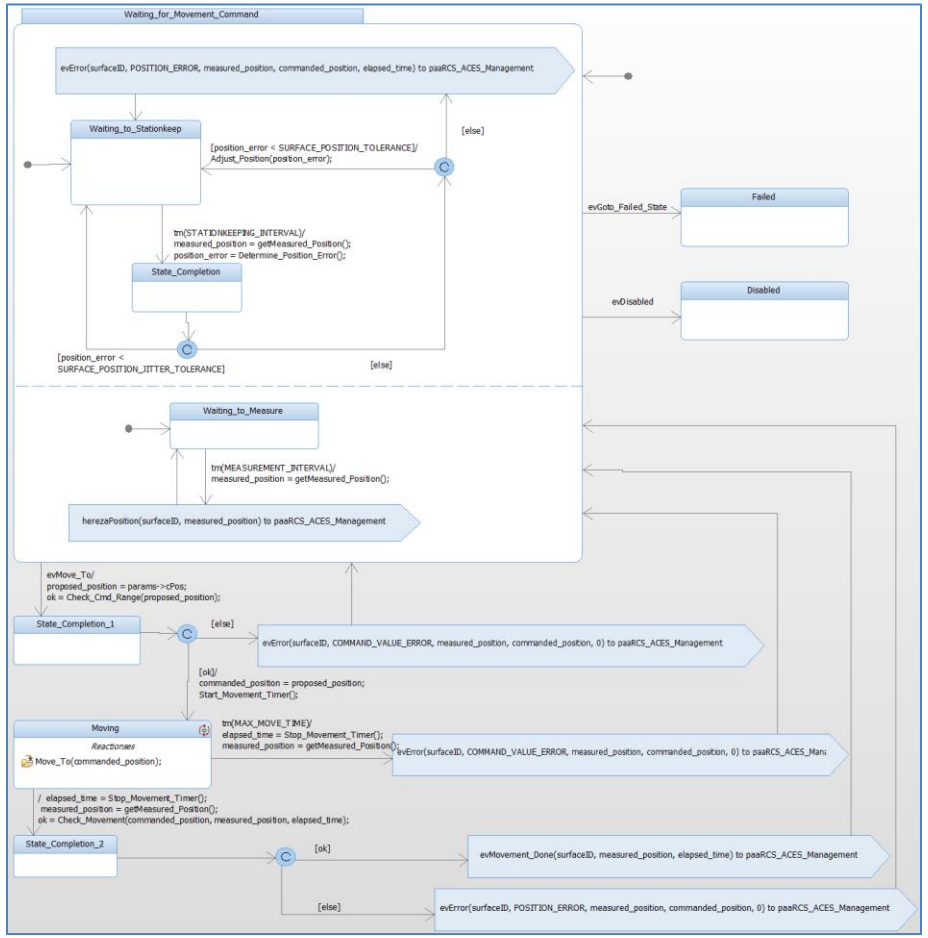
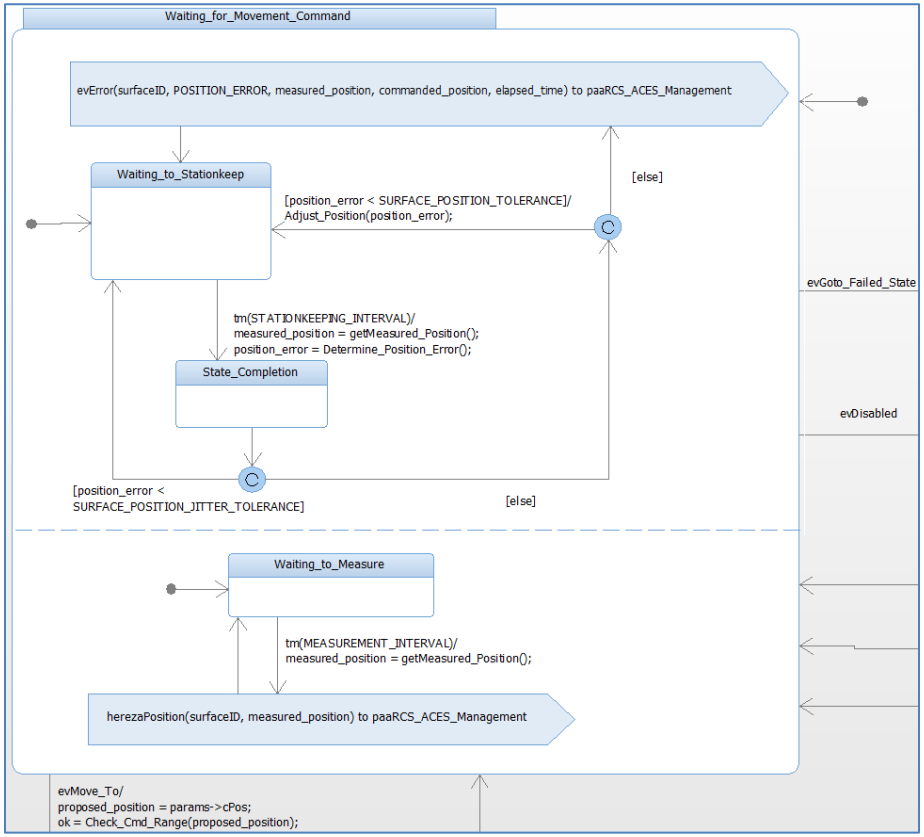
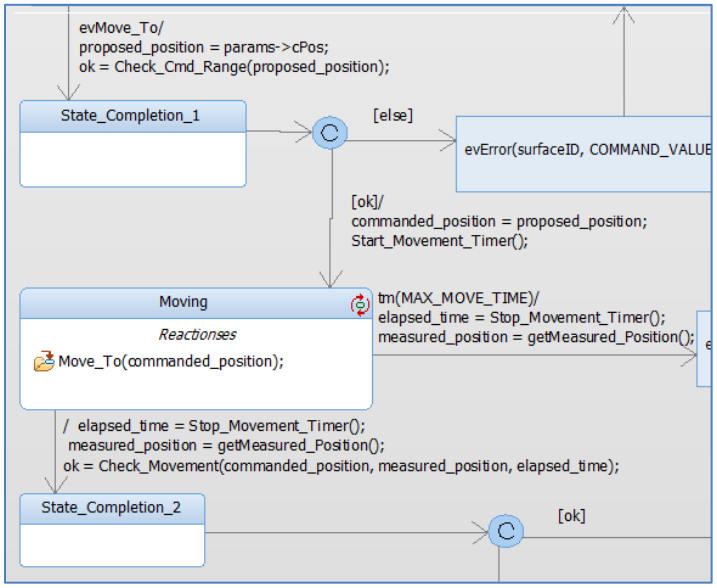


Figure 208: Rotate Control Surface use case block state machine

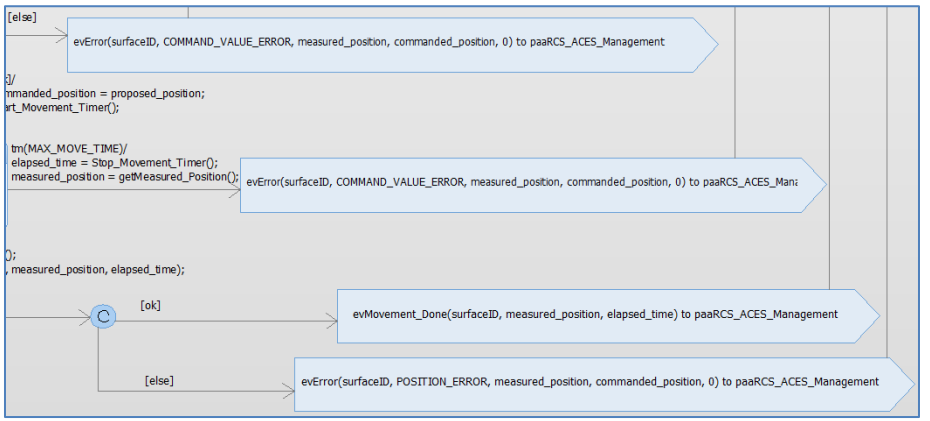
Because the image may be a bit small, here are two areas of the state machine zoomed in. First the **Waiting_for_Movement_Command** state:



and then the command processing part of the state machine:



and the part to its right:



Before this can be executed, the following elements must be defined:
Constants

- **STATIONKEEPING_INTERVAL**
- **MAX_MOVE_TIME**
- **MEASUREMENT_INTERVAL**

- **SURFACE_POSITION_JITTER_TOLERANCE**
- **SURFACE_POSITION_TOLERANCE**

The constants are just symbolic names used to represent important unchanging values.

Value Properties

- **SurfaceID surfaceID**
- **int measured_position**
- **int commanded_position**
- **int proposed_position** (we'll need this later…)
- **Second elapsed_time** (note: **Second** is defined in the SysML profile)
- **bool ok**

Value properties represent information that generally varies when the system is operational.

Operations

- **int getMeasuredPosition()**
- **void Adjust_Position(int pos)**
- **bool Check_Cmd_Range(int pos)**
- **void Start_Movement_Timer()**
- **void Move_To(int pos)**
- **Second Stop_Movement_Timer()**
- **int Determine_Position_Error()**
- **bool Check_Movement(int cPos, int mPos, Second elapsed)**

Operations, for the most part, represent system or subsystem functions that are important enough to be exposed at this level.

Defining the Constants

The constants are all relevant to the architecture and so will be stored in the **InterfacesPkg > DataTypesPkg** package. The **STATIONKEEPING_INTERVAL** is clearly needed to meet the stationkeeping requirements but its value is not specified. This means that a requirement is missing – so in a real project, we'd have to go back to the subject matter expert (or do experiments in the lab) to determine the value and add it as a requirement. For our purposes, we'll do it every 800 miliseconds so we'll define it as the value of 800. In the *ValueType* category in the **InterfacePkg > DataTypePkg**, add the item as a *Language* Kind with the declaration

```
#define %s 800
```

In similar fashion, define **MAX_MOVE_TIME** as with the value 3000 (3 seconds) and **MEASUREMENT_INTERVAL** as the value 1000.

The constant **SURFACE_POSITION_JITTER_TOLERANCE** is there to determine how big an error justifies a correction. This is not specified in the requirements, so we'll have to go back to our subject matter experts (or the lab), determine a reasonable value, and add a new requirement. For our purposes (simulation), we'll just use the value ±2:

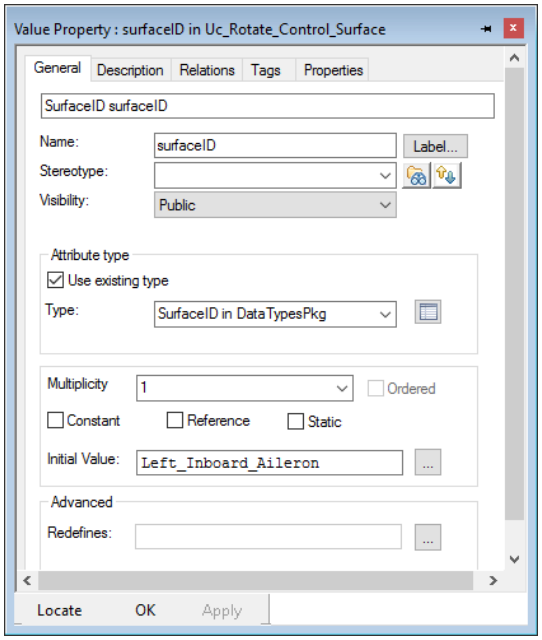
```
#define %s 2
```

The constant **SURFACE_POSITION_TOLERANCE** is a larger value that means that if the deviation is this much, then we need to raise an error. Our subject matter experts need to weigh in the the actual value we need to require but for our purposes here, we'll use ±4.

```
define %s 4
```

Defining the Value Properties

Simply add the numeric value properties to the **Uc_RotateControlSurface** use case block using the default type (*int*) and assign a initial value of zero. **ok** should be defined as a *Rhpboolean*. For the **surfaceID** value property, we'll need to type it properly and assign it an initial value. In this case, I assigned the value **Left_Inboard_Aileron**.

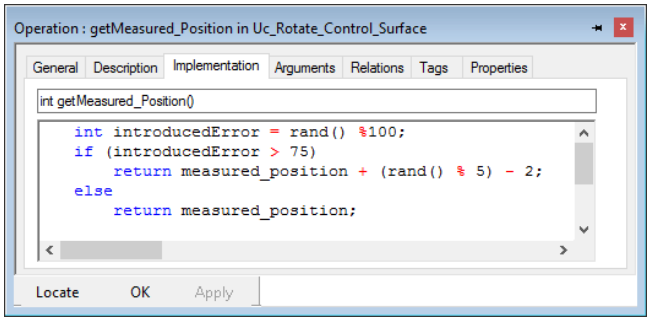


Defining the Operations

As before, it is important to remember that we're implementing these functions for the purpose of simulation support, not specifying the internal design.

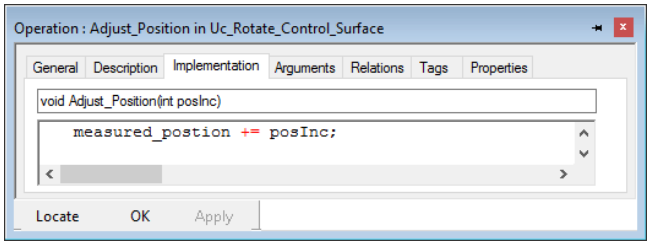
int getMeasured_Position()

The intent of this operation is to get the actual measured position. Since we're just simulating the system, here is would be usedful to add some randomness, so we'll include an implementation that adds a small random value. Sometimes it will be enough to trigger stationkeeping movement but not always.



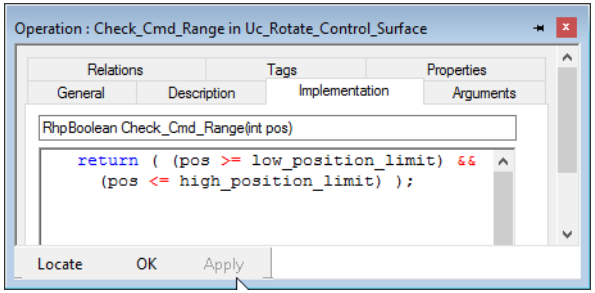
void Adjust_Position(int posInc)

This just augments the reported measured position with an offset. This is used to simulate and adjust when the measured position differs from the commanded position.



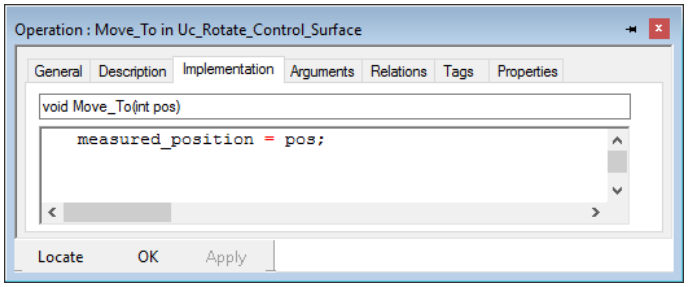
bool Check_Cmd_Range(int pos)

To add this operation, we'll also need to add two value properties to the use case block to represent the low (**low_position_limit**) and high (**high_position_limit**) set limits (actually set during configuration of the subsystem). When you define these value properties, set their initial values to -40 and 40, respectively. This function returns *TRUE* if the commanded value is within the configured limits of the control surface.

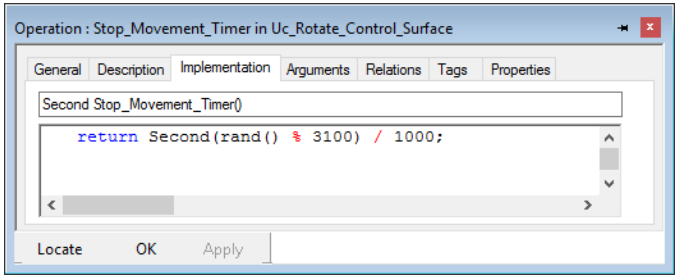


void Start_Movement_Timer()
We won't actuall time anything so this operation can have an empty implementation.

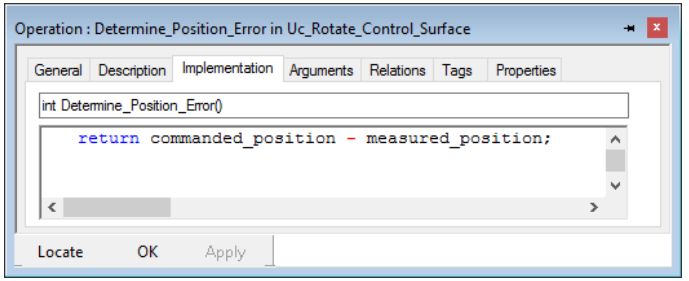
void Move_To(int pos)
This operation simulates the movement of the control position to its commanded position. Since we're not simulating the internal design, it is enough to simply assign the commanded value to the measured value.



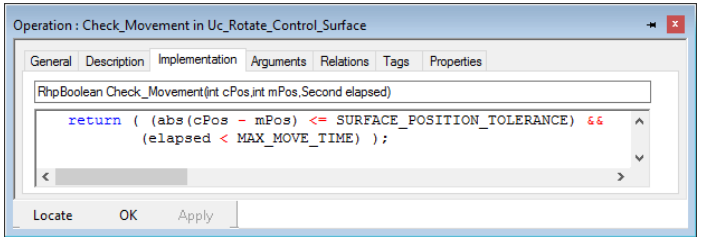
Second Stop_Movement_Timer()
This operation needs to return the time required for the movement to take place. In this situation, we'll just use a random number between 0 and 3100 and then divide it by 1000 to get the time in seconds (3.1 seconds). This means that usually it will be in range but occasionally it will not.



int Determine_Position_Error()
This function returns the error between commanded and measured positions.



RhpBoolean Check_Movement(int cPos, int mPs, Second elapsed)
This function checks the success of the movement.



Instrumenting the aaRCS_ACES_Management Actor
In this execution model, the only subsystem actor block relevant is **aaRCS_ACES_Management**, as **aaRCS_Hydraulics** and **aaRCS_Power** don't receive or emit events. Create the following state machine for the **aaRCS_ACES_Management** actor block:

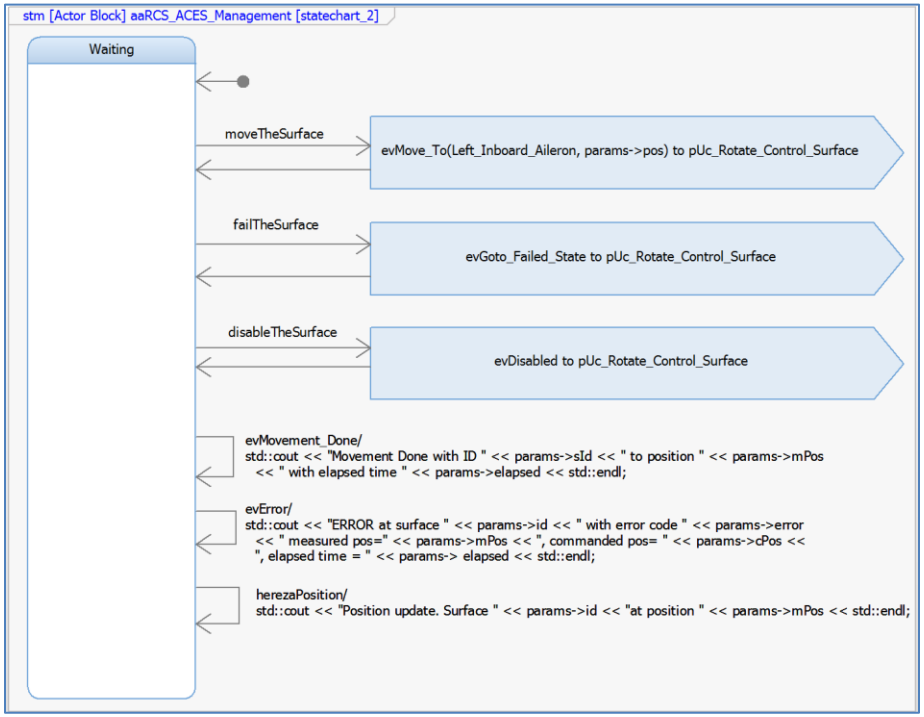
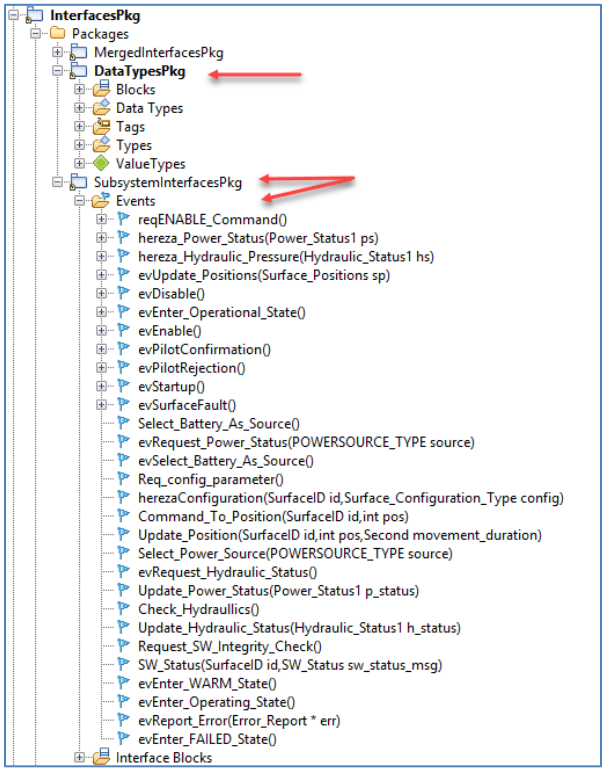


Figure 209: aaRCS_ACES_Management actor block state machine



This will enable you to run the use case state machine by driving the actors with the events **moveTheSurface, failTheSurface** and **disableTheService**.

9.3.4 Running the subsystem use case model

Compile and run the **RCS_Component::Animate** configuration that we defined in Section 9.3.2.3.

Note: For the compilation to succeed, the events defined in the **InterfacesPkg** should be in a subpackage (here there are in the **SubsystemInterfacesPkg**. They cannot be in the **InterfacesPkg** directly if the **DataTypesPkg** is a subpackage.

Here is an example output scenario, first driving the movement to position 20 (legal) to position 80 (illegal) and then to position -10 (legal). There are a number of other paths you should execute to ensure the quality of the model and its requirements.

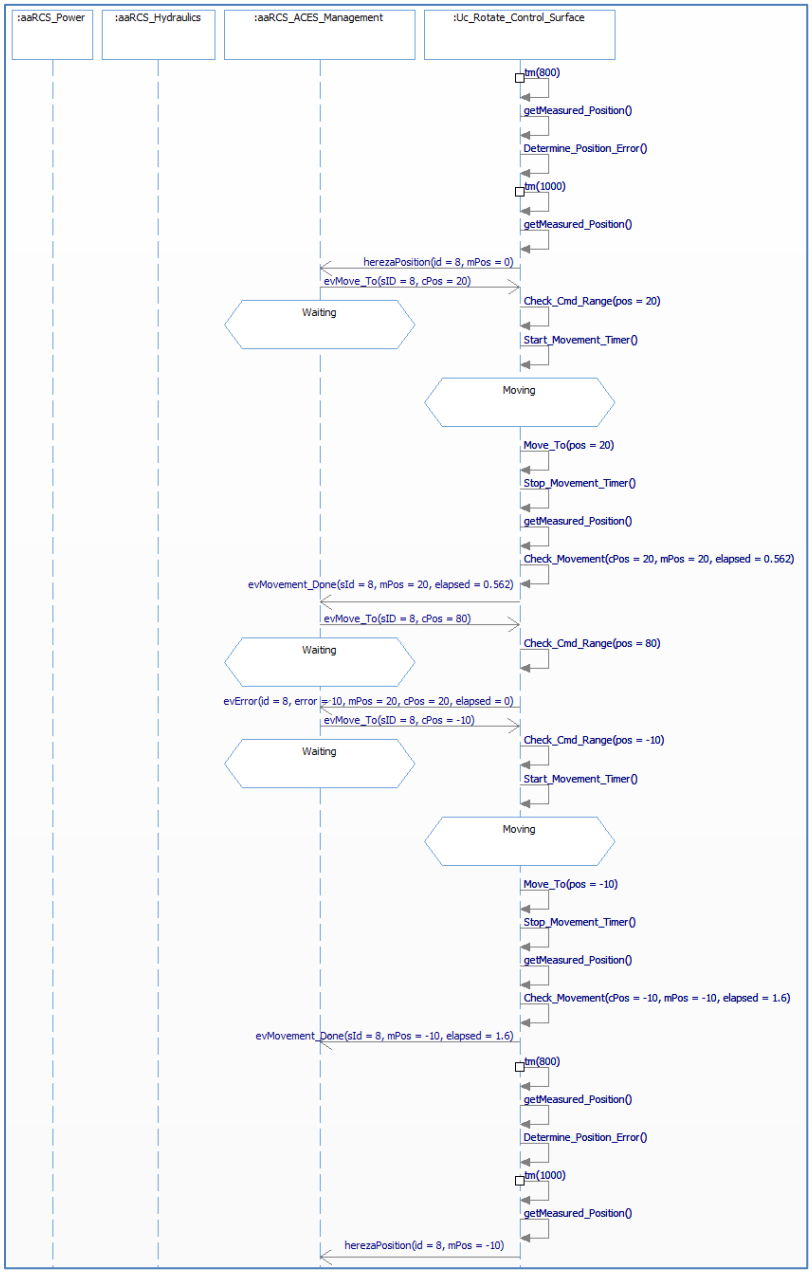


Figure 210: Sample execution of the Rotate Surface subsystem use case model

These sequence diagrams can be converted to use the actual subsystem elements using the technique outlined in Section 9.3.1.2 on page 171. To recap:

Create an appropriate package within the **WBScenariosPkg** to hold the copied sequences. Then, for each newly added sequence diagram

- Copy the subsystem use case analysis sequence diagram to the newly created package**.** Rename to add "WB" to the name to indicate it is a white box (architecture-dependent) scenario.
- Add the actual actors and subsystems to the diagram
- Retarget each local use case actor block with the actual actor or subsystem block (you can use the *SE-Toolkit > Add Subsystems* tool to assist)
- Change the source and target of the messages to reflect the real elements involved (selecting the messages and use the left and right arrow keys is the easiest way)
- If there are referenced sequence diagrams used, be sure to update the references to the copied and updated white box scenarios
- Once complete, realize the messages on the converted sequence diagram by right clicking on the diagram and selection *Auto Realize All Elements*.

If you are now going to add the scenarios from the **Rotate Control Surface** use case model, create the package **DesignSynthesisPkg > ArchitecturalDesignPkg > WBScenarios > ControlAirSurfacesWBScenariosPkg** to contain them (since the subsystem **Rotate Control Surface** use case is derived from the system level use case **Control Air Surfaces**).

If we do this for the sequence produced for the **Rotate Control Surface** use case, we get a **WBScenariosPkg** structure that looks like this:

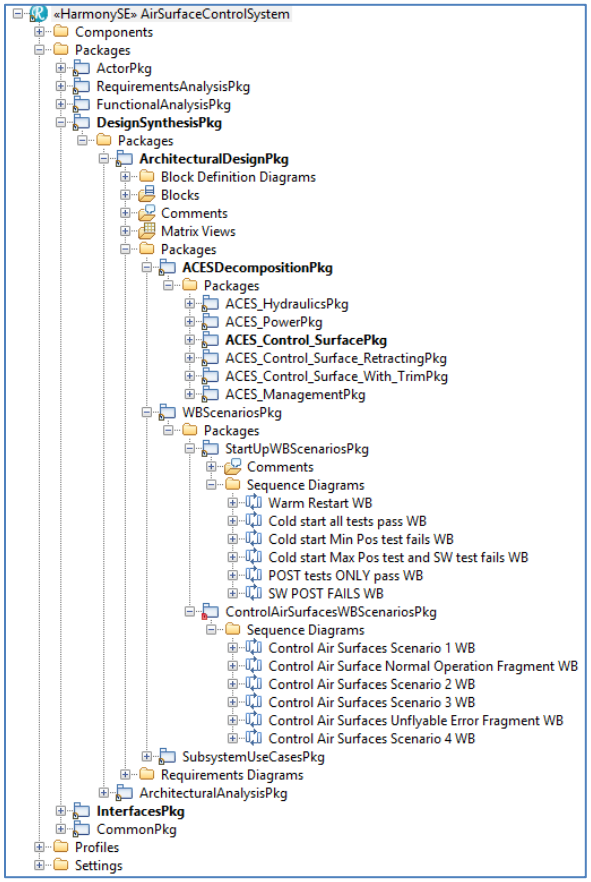


Figure 211: White box architectural scenarios

Here is a white box architectural version of scenario 1 of the system use case Control Air Surfaces. Compare to Figure 120 on page 94.

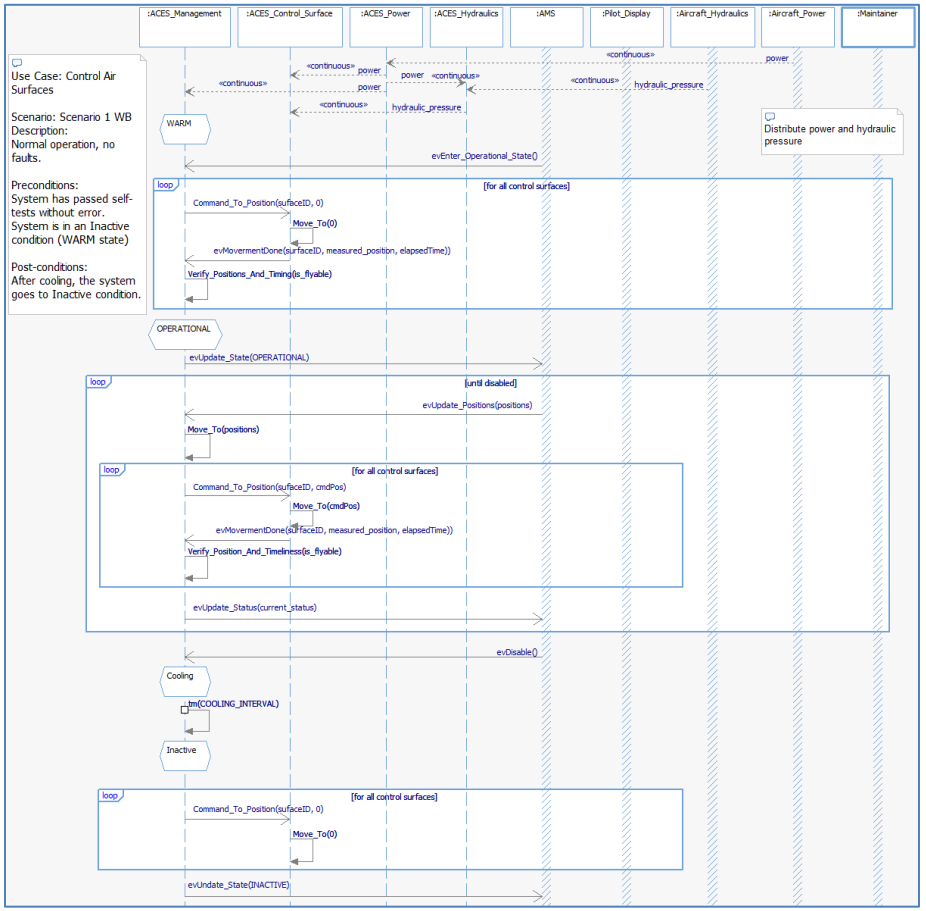


Figure 212: White box version of Control Air Surfaces Scanario 1

Next, we can repeat this process for Scenario 2:

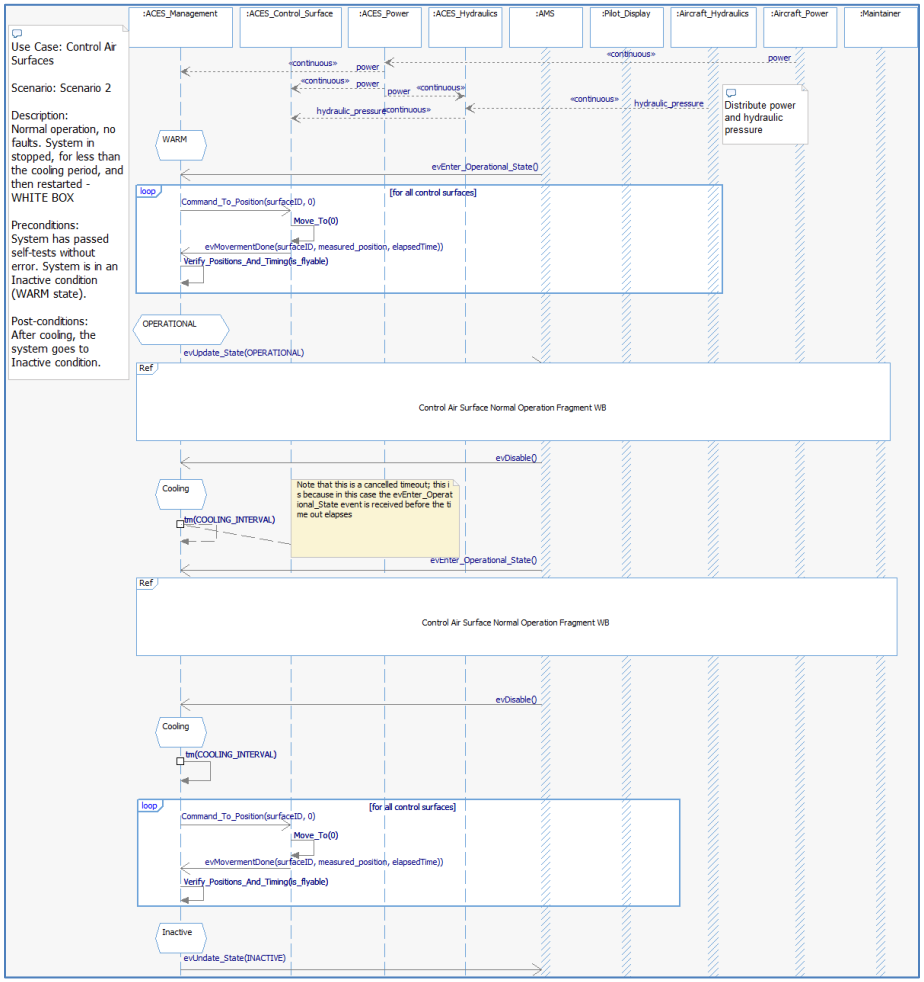


Figure 213: White box version of Control Air Surfaces Scenario 2

The white box version of the normal operation interaction fragment is shown below:

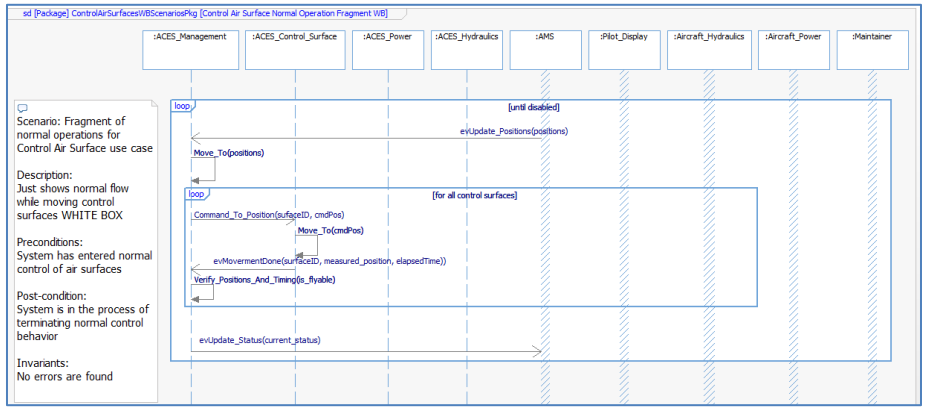
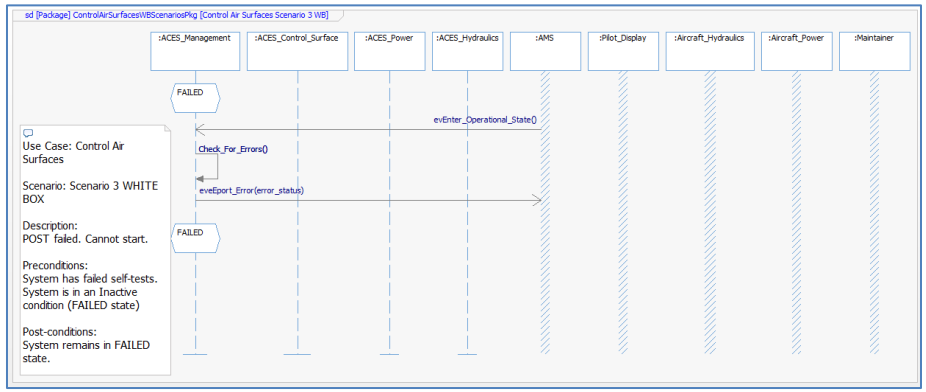


Figure 214: White box version of the Normal Operation interaction Fragment

The white box version of Scenario 3 is simple:



The last scenario is Scenario 4 in which unflyable errors are discovered.

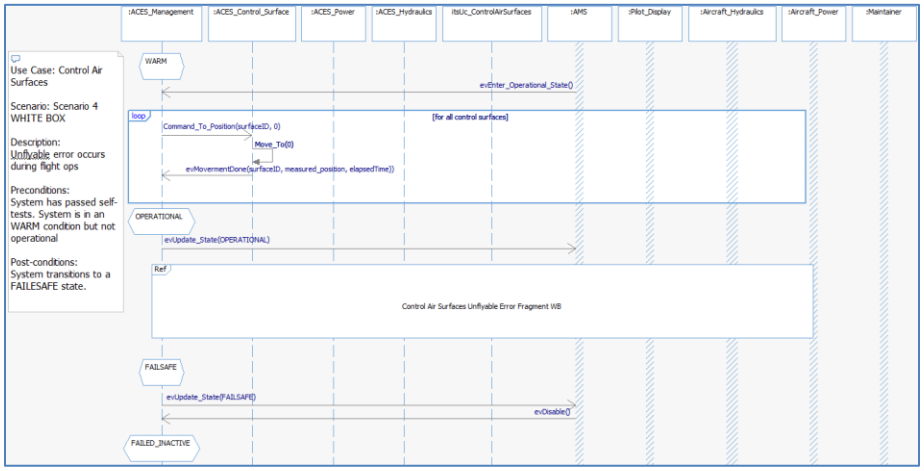


Figure 215: White box version of Scenario 4

Finally, the details of the white box version of the unflyable interaction fragment are shown in Figure 216.

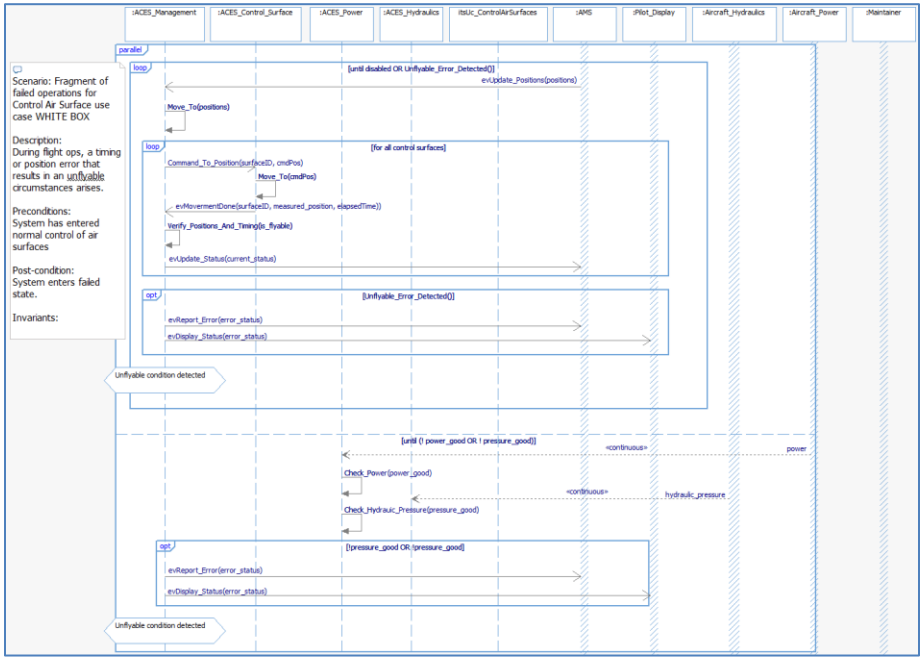


Figure 216: White box version of the unflyable interaction fragment

You will also need to do this for any subsystem use cases case well. Here is the white box architectural version of the **Rotate Control Surface** scenario 3 from Figure 206. This replaces the stand-in actors used for simulation purposes with the actual actors.

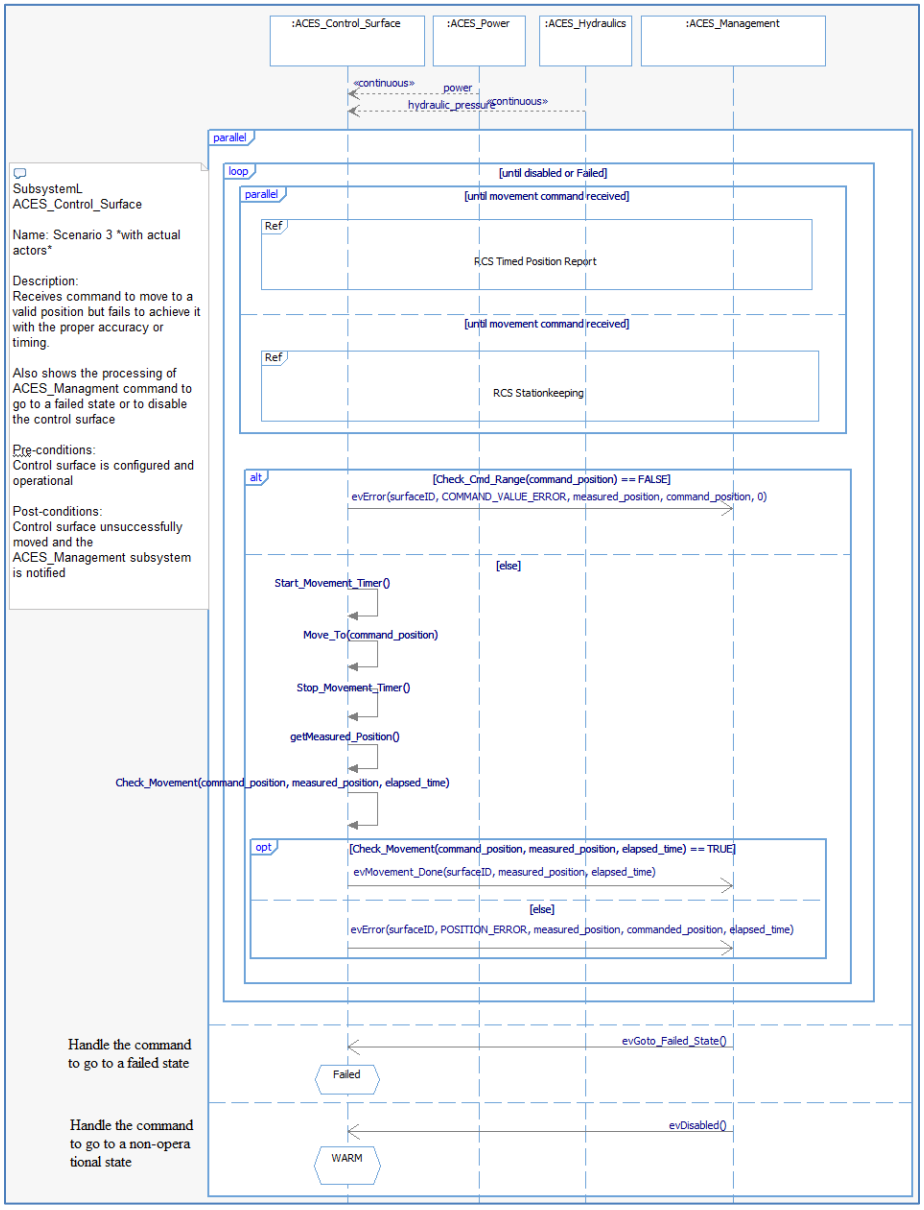


Figure 217: White box Rotate Control Surface use case scenario 3

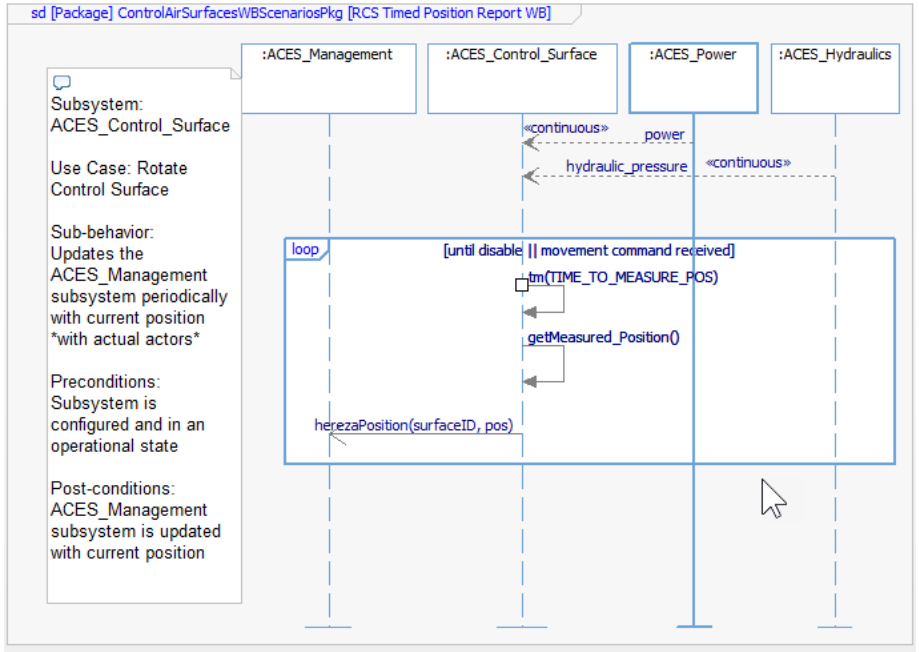


Figure 218: Architectural version of Timed Position Report interaction fragment

The similarly updated referenced interaction fragments are shown below.

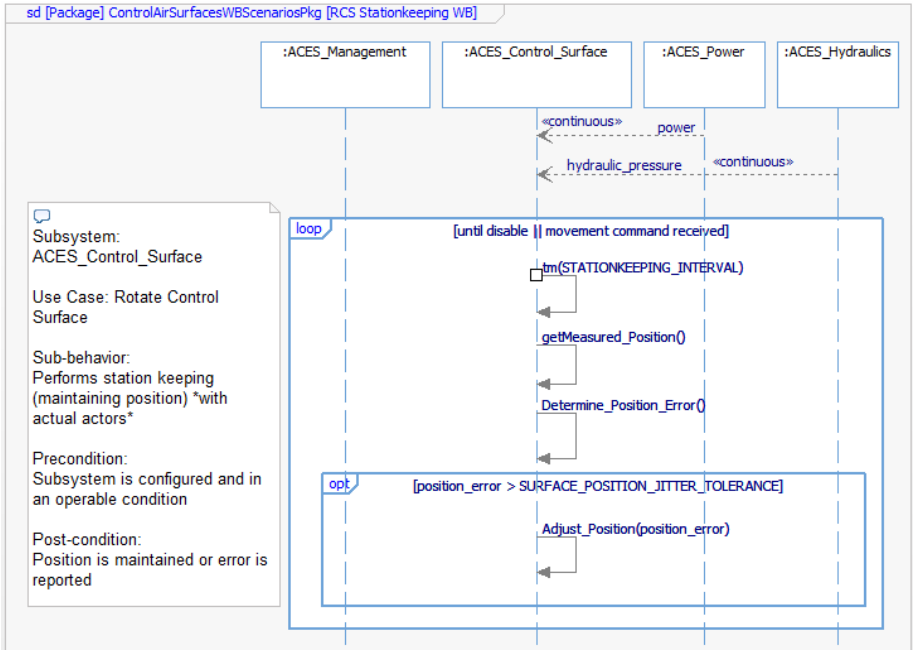


Figure 219: Architectural version of the Stationkeeping interfaction fragment

Note that the lifelines are the actual subsystems and not the original local stand-ins used for the functional analysis of the subsystem use case. Also notice that the referenced sequence diagrams on Figure 217 reference the newly created and modified copies of those interaction fragments, not the original.

9.4 Create/Update Logical Data Schema

In this task, we will be creating the architectural data and flow schema for the architecture. This schema must take into account all the analyzed use cases as well as the specified architecture. This step is crucial because it will be a hugely important input into the definition of the system interfaces, performed in Section 9.5.

The good news is that many of the data types can just be copied, renamed and used from these previous analyses. The bad news is that we cannot just reuse the data schema diagrams, as they will refer to the original (and use-case specific) types. These diagrams must be manually recreated in the project-level **TypesPkg**. In addition, as we define additional use cases, we are likely to identify data types that must be (manually) merged because they must take into account multiple use cases and additional requirements.

Reusing System Functional Analysis Types

First, let us consider the types that can be directly reused from the system functional analysis. The previoius task of merging the functional analysis moved many – if not all – of the types.

As a default, you can create diagrams with the same organization of elements as the original data schema diagrams (as block definition diagrams, of course) but using the new system types instead of the original types from the functional analysis. Compare Figure 128 with Figure 220, below.

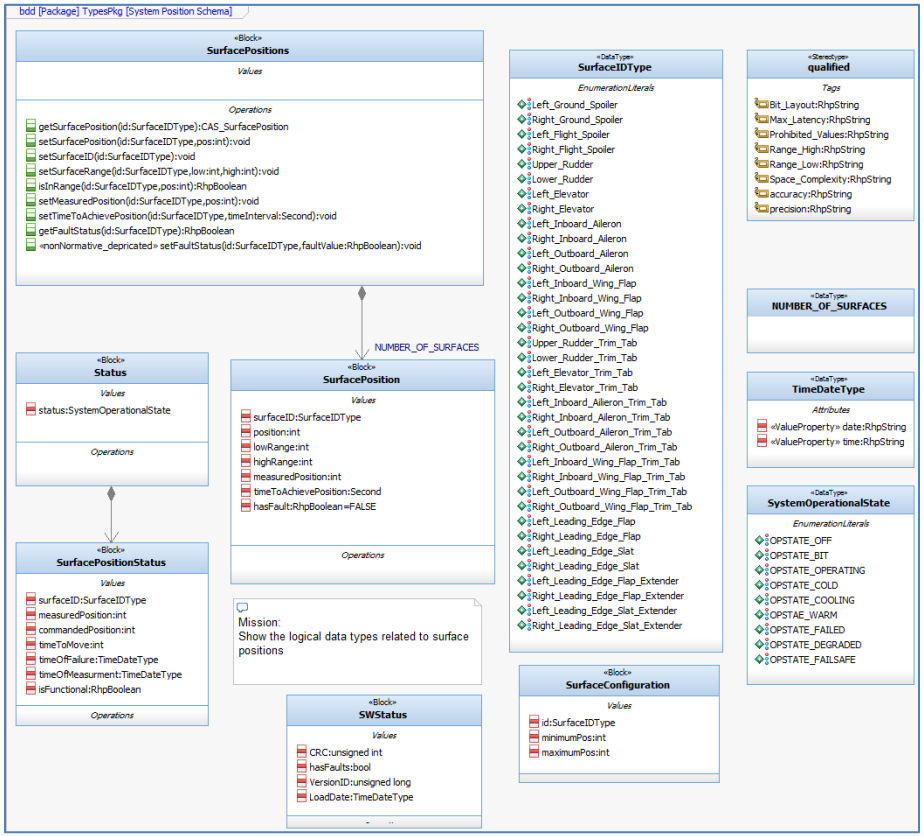


Figure 220: System Data Schema for control surface positions

We must also replicate the **Start Up Data Schema** diagam (Figure 80) using the elements in the **InterfacesPkg > DataTypesPkg** (Figure 221). Again, this means updating all references to the types used in the use case functional analysis and replacing them with references to their counterparts in the **DataTypesPkg**. This includes the types referenced with relations (composition and dependency in this case) and the types used to specify the attributes and value properties.

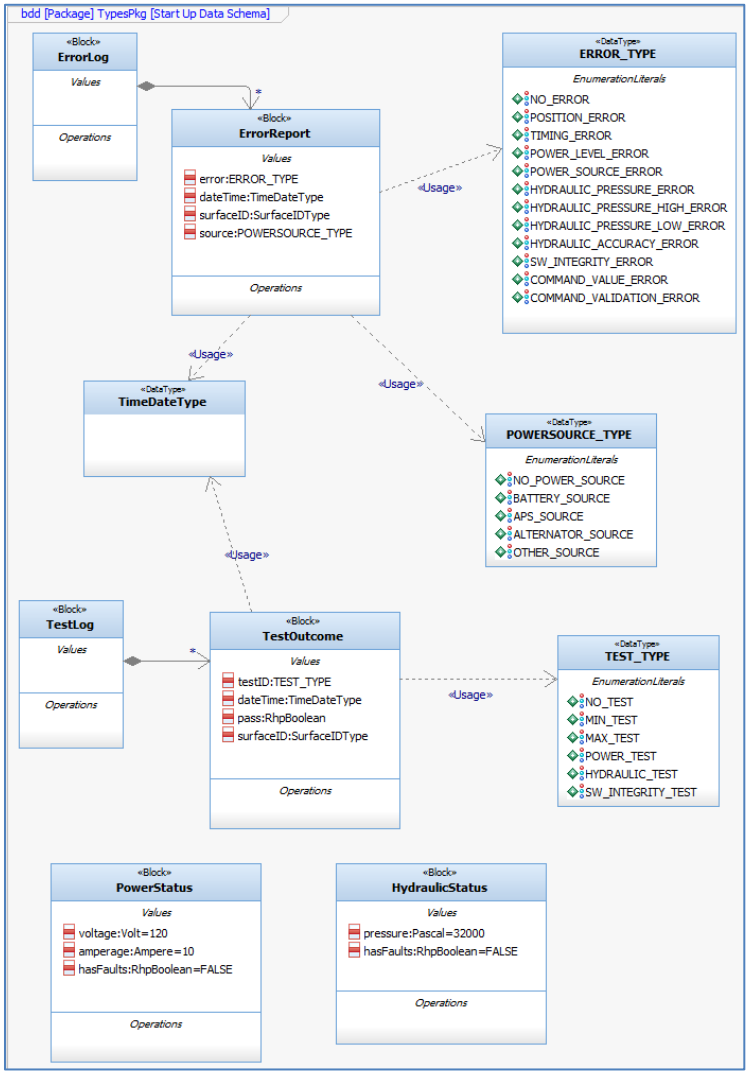


Figure 221: Start up use case architectural type schema in TypesPkg

Reusing Types from Subsystem Use Case Analyses

Now, let's look at the types who structure and content were identified during the analysis of subsystem use cases.

In this case, no new types were identified in the subsystem use case analysis no nothing must be added for the schema.

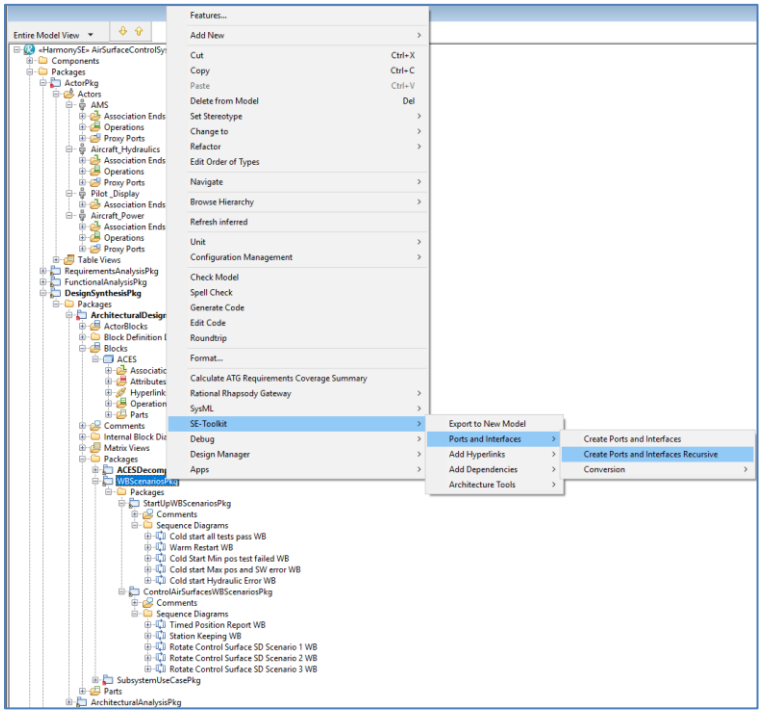
9.5 Define / Merge System Logical Interfaces

System logical interfaces include both subsystem-actor and subsystem-subsytem interfaces. These will ultimately come from the system use case functional analysis for use cases not decomposed, or from the detailed analysis of the subsystem use cases for those use case which are decomposed. These are the *logical interfaces* between these contextual or architectural elements and will be captured as interface blocks. Physical interfaces will be derived from these in the Handoff Worflow, which is described in detail in Section 10.

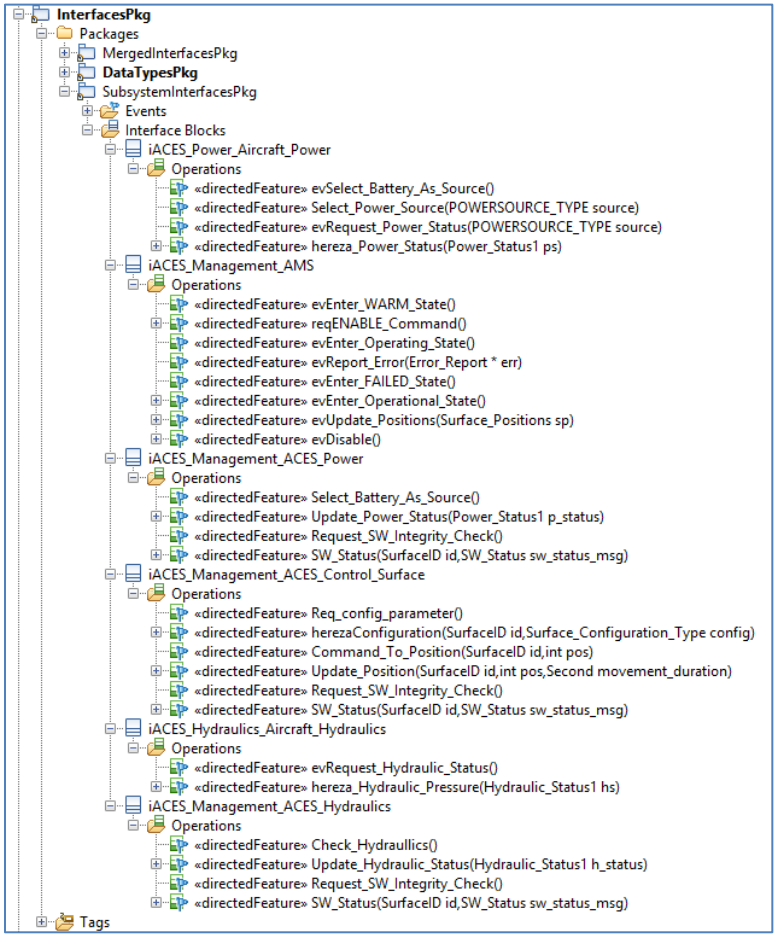
If we've done a good job in defining the white box scenarios, then they contain all the information we need. It is important to not only identify the events that get passed around the archtictural element; we must also identify and characterize the data they carry and any separate flows not carried by events. This means that it is crucial that this information be provide in the sequence diagram and the types (identified in the previous section) be fully specified in all their logical glory.

Note that the *Create Ports and Interfaces* tool creates the events as directed features in the interfaces but does not necessarily include all the types (depending on how the sequences from which the interfaces were derived were created). You must review these identified services thorough to ensure they are complete with data types.

- Right click on the **DesignSynthesisPkg > ArchitectureDesignPkg > WBScenariosPkg** and select *SE-Toolkit > Ports and Interfaces > Create Ports and Interfaces Recursive*



- Look at the created interface blocks in the **InterfacesPkg**:



- Walk through these interface blocks and ensure that for each service the parameter list matches the sequence diagram call. For example, in the figure above the following events are missing parameters that will need to be added (by editing the referenced events (not the event receptions)). You can find the appropriate parameter lists be going to the originating white box scenarios in the **WBScenariosPkg**. This must be done for all interface features generated using the "bottom up" approach discussed earlier. You may find that you need to create new types such as **SurfaceConfiguration** and **SWStatus**.
  - **iACES_Management_ACES_ControlSurface**
    - **Command_to_Position()**
    - **Updated_Position()**
    - **herezaConfiguration()**
    - **SW_Status()**
  - In the case of the **Command_To_Position()** event reception, the following parameters should be added:
    - **id: SurfaceIDType**
    - **pos: int**
  - Also identify misspellings and merge together any features that are synonymous. For example in the **iACES_Management_ACES_ControlSurface** there is both a **Command_To_Position()** and **Command_to_Position()** event reception that differ only in the case of the **_to_** part of the name. These are clearly meant to be the same. Delete the one with the lower case "**_to_**". Also delete the corresponding event reception from the **ACES_ControlSurface** subsystem block.

- Add any flows on the diagrams as flow properties to the appropriate interface blocks. In the case, add the following flow properties
  - A flow named **power** from the **ACES_Power** subsystem to the **ACES_ControlSurface** subsystem,defined as of type *Ampere* (from the SysML profile).
    - Add this to the **iACES_Management_ACES_Power** interface block.
    - Stereotype this flow as a **«directedFeature»** with a direction of **in**.
    - Add the flow property to both the **ACES_Power** and **ACES_ControlSurface** subsystems
  - A flow named **hydraulic_pressure** from the **ACES_Hydraulics** subsystem to the **ACES_ControlSurface**

subsystem defined as being of type *Pascal* (from the SysML profile).

- Add this to the **iACES_Management_ACES_Hydraulics** interface block.
- Stereotype this flow as a **«directedFeature»** with a direction of **in**.
- Add the flow property to both the **ACES_Hydraulics** and **ACES_ControlSurface** subsystems

- Add a flow from the actor **Aircraft_Power** to the **ACES_Power** subsystem.
  - Name this flow **power**.
  - Add it to the **iACES_Power_AircraftPower** interface block.
  - Stereotype this flow as a **«directedFeature»** with a direction of **in**.
  - Add the flow property to the **Aircraft_Power** actor
- Add a flow from the actor **Aircraft_Hydraulics** to the **ACES_Hydraulics** subsystem.
  - Name this flow **pressure**.
  - Add this flow to the **iACES_Hydraulics_Aircraft_Hydraulics** interface block.
  - Stereotype this flow as a **«directedFeature»** with a direction of **in**.
  - Add the flow property to both the **Aircraft_Hydraulics** actor
- Add a new block diagram to add the new ports connecting the **ACES_ControlSurface**, **ACES_Power** and **ACES_Hydraulics** subsytems. This is because the *Create Ports and Interface* wizard did not create these interface because there is only flows between these subsystems and no events.

- In the **DesignSynthesisPkg > ArchitectureDesignPkg > ACES_DecompositionPkg** add a new block definition diagram name **ACES Flow Connections**
- Drag the **ACES_ControlSurface**, **ACES_Power** and **ACES_Hydraulics** onto the diagram
- Add proxy ports and interface blocks, as shown in Figure 222

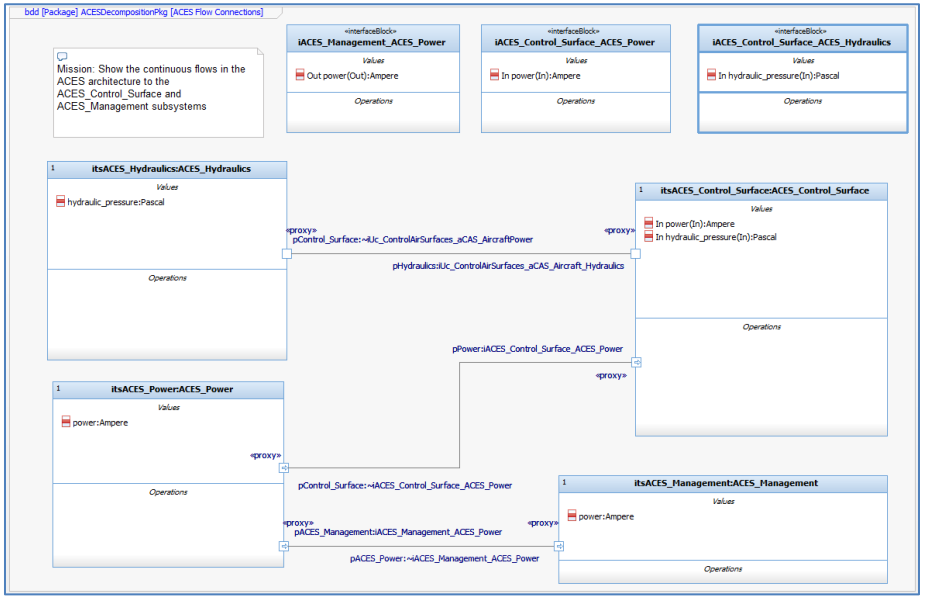


Figure 222: Architectural Flow interfaces and Connections

- Move (not copy) the newly created interface blocks from the **DesignSynthesisPkg > ArchitectureDesignPkg > ACES_DecompositionPkg** package to the **InterfacesPkg.**

Once this is all done, the updated interface blocks features should be updated with parameters and types and look like Figure 223.

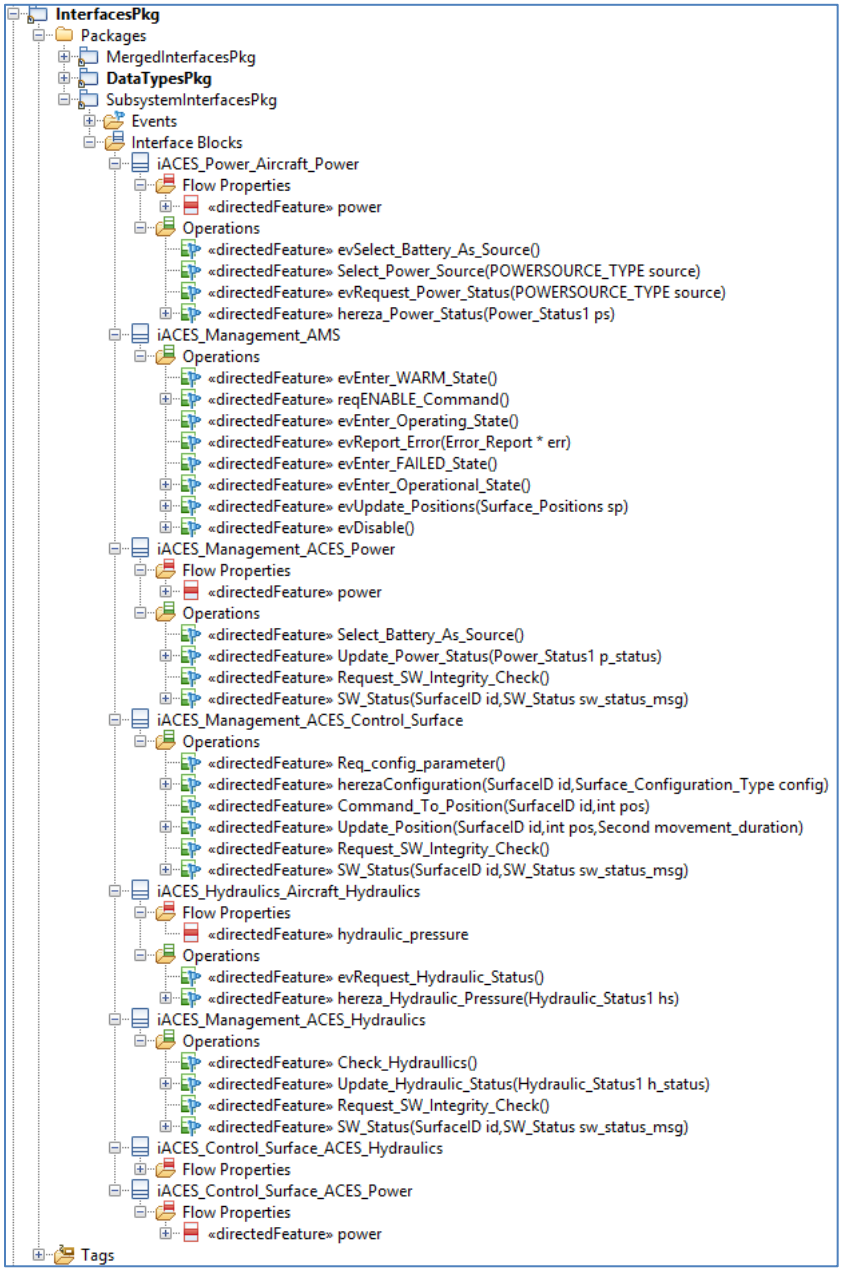


Figure 223: Interface Blocks updated with parameters and types

9.6 Analyze Dependability

We won't perform this activity in this Deskbook in order to keep it a little shorter. We'll just say a three of things about it here.

First, as we've mentioned before, dependability analysis is an ongoing parallel activity to requirements and design. It ensures that the created engineering data and work products meet the safety, reliability, and security needs of the customer. As we make design decisions – and architecture is heavily design focused – we introduce the possibility that 1) we didn't properly address concerns already identified, and 2) we introduced new concerns. Therefore, as we define and evolve the architecture, we must maintain and update our dependability analyses.

Secondly, if you're using Rhapsody to perform such dependability analysis, then you need to have a place to put it. Previously, we added a package for each use case in the functional analysis package to hold all use case detail. We added subpackages to organize this detail, including a Safety Analysis package. We will do the same here. Since this is a type of architectural analysis, we'll add it into the **ArchitecturalAnalysisPkg** Package as shown in Figure 224.

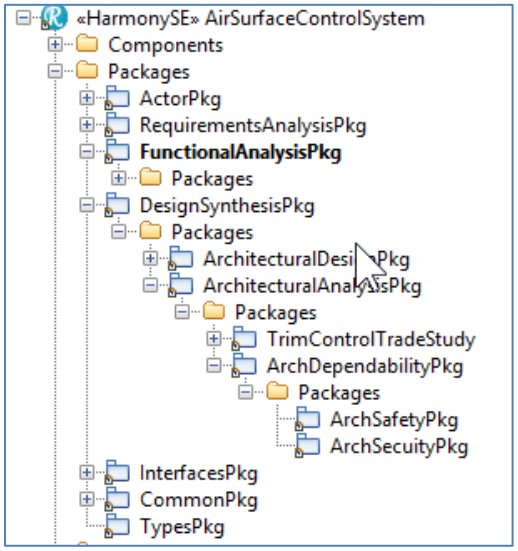


Figure 224: Architectural Dependability Analysis

Thirdly, the result of such analysis at the architectural level is usually to generate more requirements. These requirements are due to the interaction of the existing safety requirements and the addition of design and technology decisions. These newly identified requirements are allocated to subsystems and result in work in the parallel activities of *Create/Update Subsystem Requirements* and *Allocate Use Cases to Subsystems* (see Figure 10 on page 19).

10 Case Study: Handoff to Downstream Engineering

The purpose of the *Handoff to Downstream Engineering* is to

- Refine the system engineering data to a form usable by downstream engineers
- Create separate models to hold the prepared engineering data in a convenient organizational format
- For each subsystem, work with downstream engineering teams to create a deployment architecture and allocate system engineering data into that architecture

It is crucial to understand that the handoff is a *process* and not an *event*. There is a non-trivial amount of work to do to perform the above objectives. As with other activities in the Harmony aMBSE process, this can be done a single time, but is recommended to take place many times, in an iterative, incremental fashion. It isn't necessarily difficult work, but it is necessary work for project success.

The refinement of the systems engineering data is necessary because to this point it has been primarily focused on its conceptual nature and logical properties. What is needed by the downstream teams are the physical properties of the system – along with the allocated requirements – so that they may design and construct the physical subsystems.

The workflow for this activity is shown in Figure 13 on page 22 but is replicated below in Figure 225.

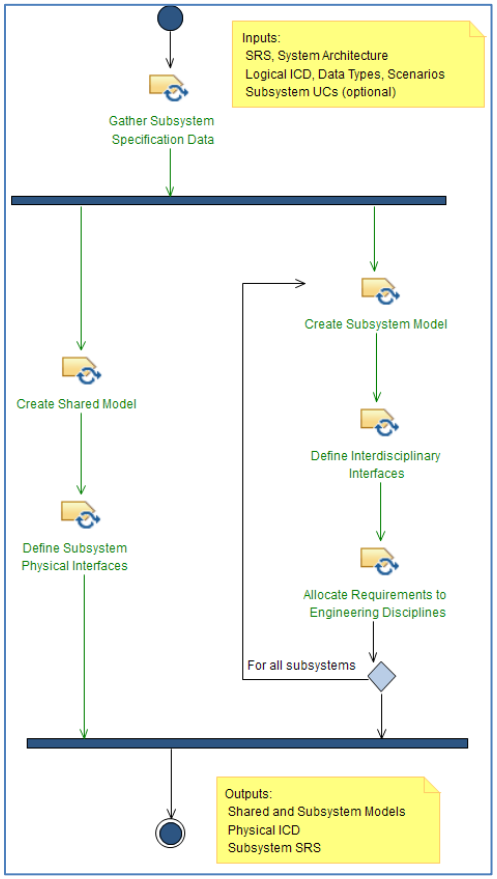


Figure 225: Handoff Workflow

10.1 Gather Subsystem Specification Data

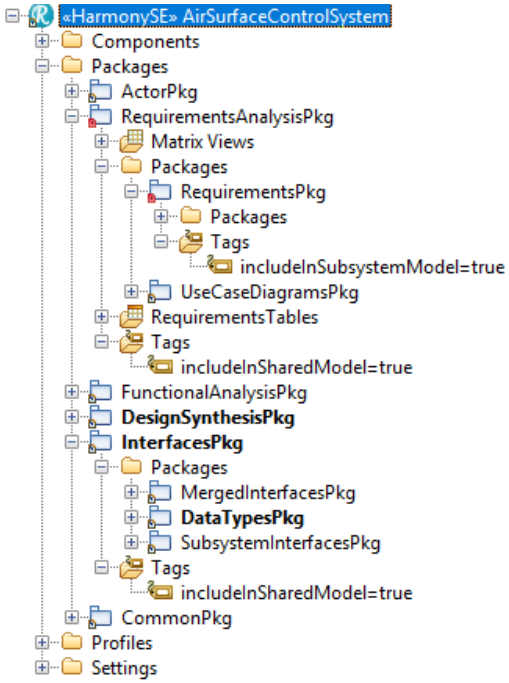
This task refers to the gathering together of the information to support the hand off. However, if you've organized the model how the Deskbook recommends, it's already done! Good for you.

10.2 Create the Shared Model

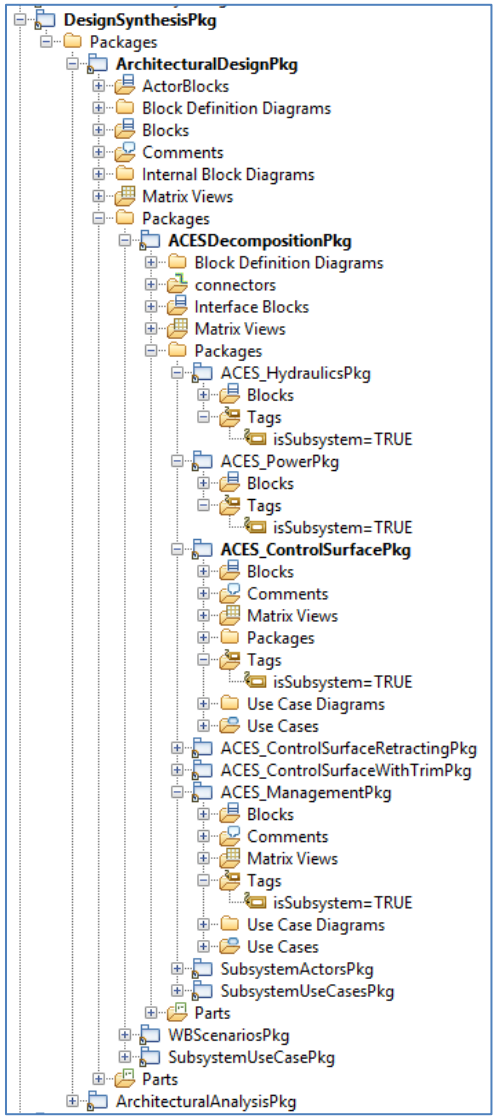
The SE Toolkit can create the basic struture of the **Shared** and subsystem models for you. The tool kit provides automation here and the created

models follow the recommended model structure supported by the SE Toolkit.

- To be included in the automatic creation of the **Shared** model, packages must be tagged with **includeInSharedModel** (of type *Boolean* with the value set to **TRUE**).



- Mark the following packages with the **includeInSharedModel** tag
  - **RequirementsAnalysisPkg**
  - **InterfacesPkg**
  - **TypesPkg**
- To be create a relevant subsystem model for hand off each of the relevant subsystem packages must be marked with the tag **isSubsystem** (of type *Boolean* with the value set to **TRUE**).



- Mark the following packages with the **isSubsystem** tag
  - **ACES_HydraulicsPkg**
  - **ACES_PowerPkg**
  - **ACES_ContolSurfacePkg**

- **ACES_ManagementPkg**

- In the browser, right click on the project (at the top) and select *SE-Toolkit > Architecture Tools > Create Handoff Models*.

This results in the creation of the shared and subsystem models. A project set is then loaded into Rhapsody with the system engineering model and the created models as separate projects. This is to facilitate the setting of properties. The models can still be loaded and worked on separately, as desired. Figure 226 shows the starting model organization for the the **Shared** model and one of the subsystem models (the other subsystem models are organized similarly).

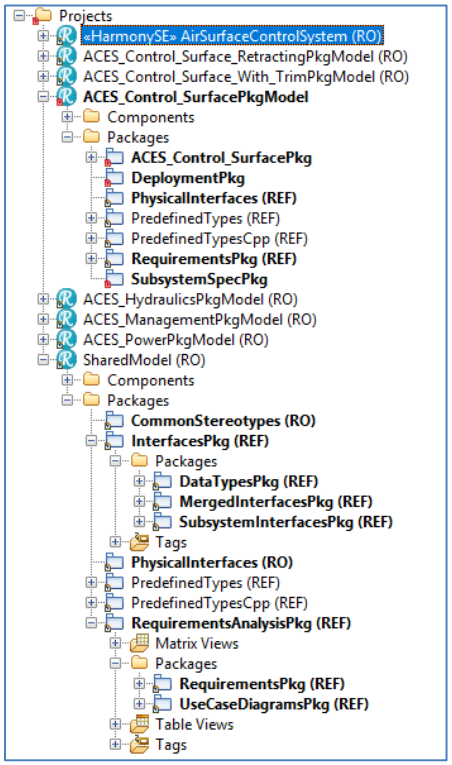


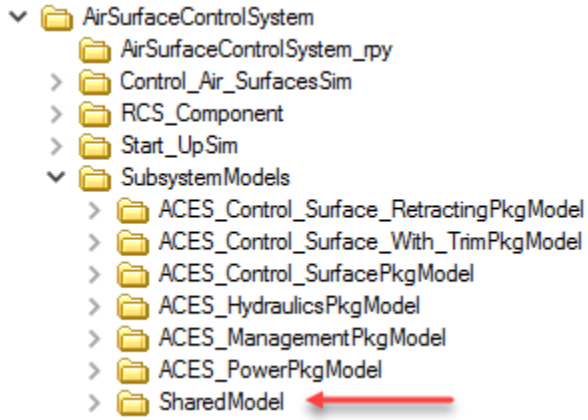
Figure 226: Models created with Create Handoff Models tool

You should also note these the created models are UML models, rather than SysML. This is because we anticipate that a great deal of the downstream work will proceed in software. Nevertheless, if desired (and we'll see later why it might be, you can always add the SysML and HarmonySE profiles.

10.2.1 Define the Physical Interfaces

The primary purpose of the **Shared** model is to contain elements relevant to multiple subsystems. This includes the physical interfaces between architectural elements and common physical types passed by those interfaces.

- At this point, close Rhapsody with the project list and open the Shared model which is located in the file system as a folder under the folder containing the SE project:



Interface blocks inherently contain specifications of services or flows. So far, all the interface blocks in the **InterfacesPkg > SubsystemInterfacesPkg** package (contained by reference in the Shared model) specify services with event receptions, which may or may not carry data. These serve as the *logical specification* of the interfaces. However, the subsystem teams are going to do detailed design and implementation of actual, physical subsystems and must use the physical interfaces of those systems. The

purpose of this task is to derive the physical interface specifications from the logical ones.

This task produces two related, but distinct work products. The first is the specification of the actual interfaces, whether they be message passing, protocol-oriented, electrical, or mechanical. That is the subject of this section. The second part is the specification of the physical data schema which includes physical details of the data, such as bit mapping of values. This latter work product will be the subject of Section 10.2.2.

Three distinct interfaces are going to be used in this system. The first is the messaging interface which includes an electrical and software protocol specification, together known as the **Control Bus Protocol** (**CBP**). Then there are also the power and hydraulic interfaces. The following sections will discuss these in detail.

10.2.1.1 Control Bus Protocol

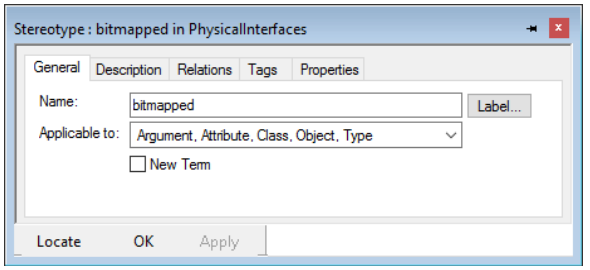
For most of the interfaces, the system will use a custom communications protocol known as the Control Bus Protocol which runs on top of an RS-232 physical electronics layer. Since the RS-232 electronic specification is available elsewhere (see, for example https://en.wikipedia.org/wiki/RS-232), we will focus exclusively on the software aspects of the protocol. Almost all the services currently defined as events in the interface blocks will be refined from this logical realization to a physical message implementation. We will do that in this section.

Useful Stereotypes

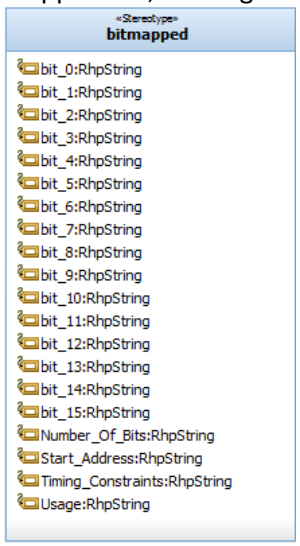
The following stereotypes are used to formally specify the physical message schema. These all the specification of the actual bit and byte structure of the message features. These stereotypes should be added to the PhysicalInterfaces package so that they are visible to the subsystem models.

«bitmapped»

This stereotype is used for value properties/attributes, variables, or registers that use bit fields to represent information.

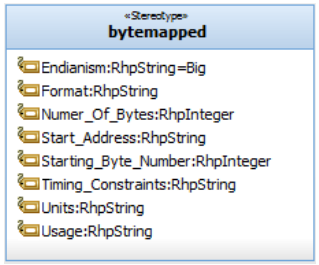


It has tags that allow the specification and usage of the bit fields, including, if applicable, starting address in a memory map.



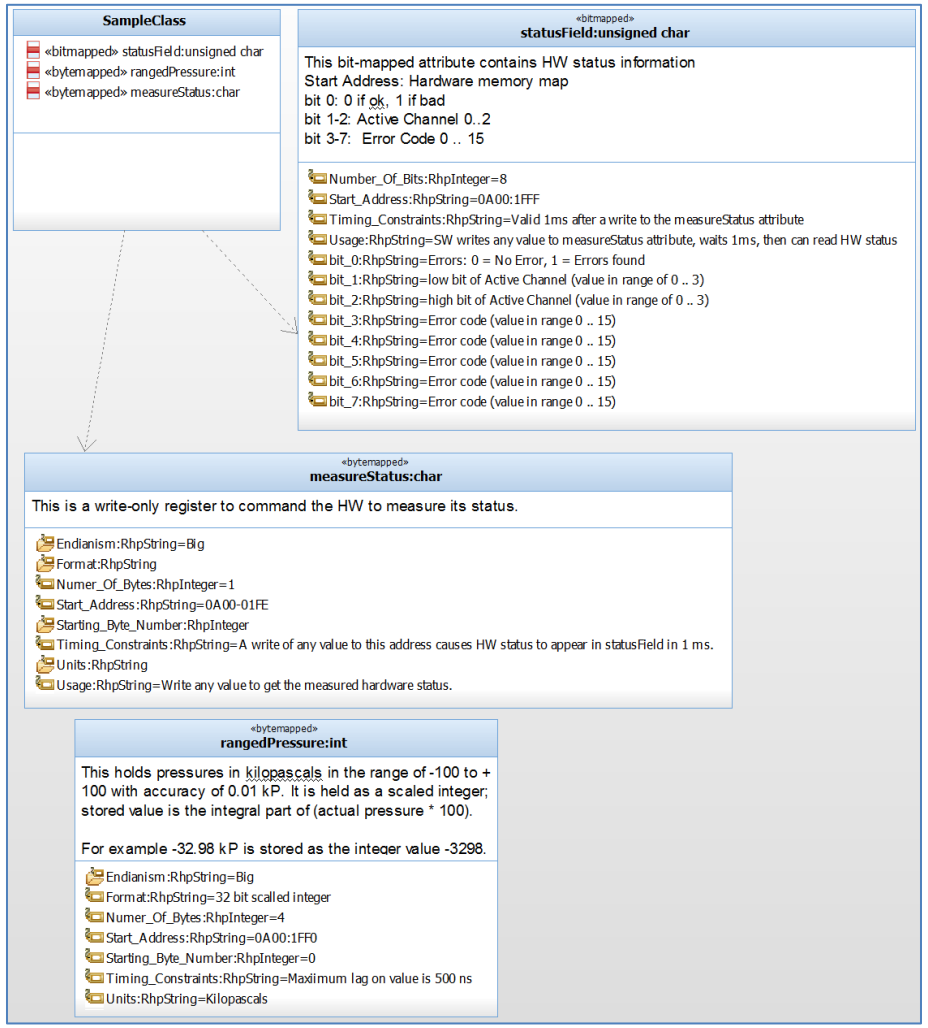
«bytemapped»

This stereotype is used for value properties/attributes, variables, and registers that are byte-mapped, including, if applicable, start address in a memory map.



Example

Below, I've shown SampleClass that has three attributes. **statusField** is bit mapped with 3 fields held in bit combinations. **measureStatus** is a hardware register in a memory map (at address 0A00-01FE) that is 1 byte in size and is a write-only hardware register. **rangedPressure** is an example of a ranged real value, whose valid range is -100.00 kP to +100.00 kP, but is held as a scaled 32-bit integer value. The stored value is 100 times the actual value and only the integral part is stored. Thus, an actual value of -32.98 kP would be stored as an integer value of -3298.



Defining the protocol

- Add two packages to the **PhysicalInterfacesPkg**.
  - **PhysicalTypesPkg** will hold base types used by the protocol messages

- **MessageTypesPkg** will hold the message definitions

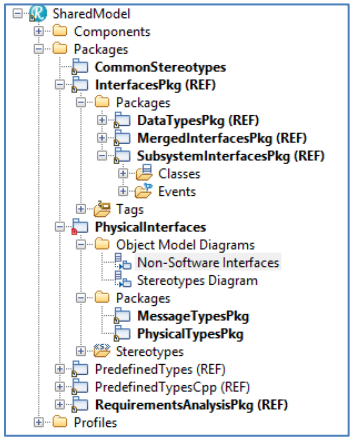


Figure 227 shows the base structure for a CBP message. All of the fields are stereotyped as **«bytemapped»** so we can define the size (in bytes), it's position in the message, whether it is big- or little-endian, and define its usage. In addition to this more formal specification of the bit format of the message, the diagram contains a comment that summarizes the structure.

The command byte is 2 bytes (16-bits) in size, big-endian format and holds one of the values of the **CBP_Command** enumerated type, shown at the right of the figure. Most of these messages will have actual content fields, which are defined in the relevant subtype.

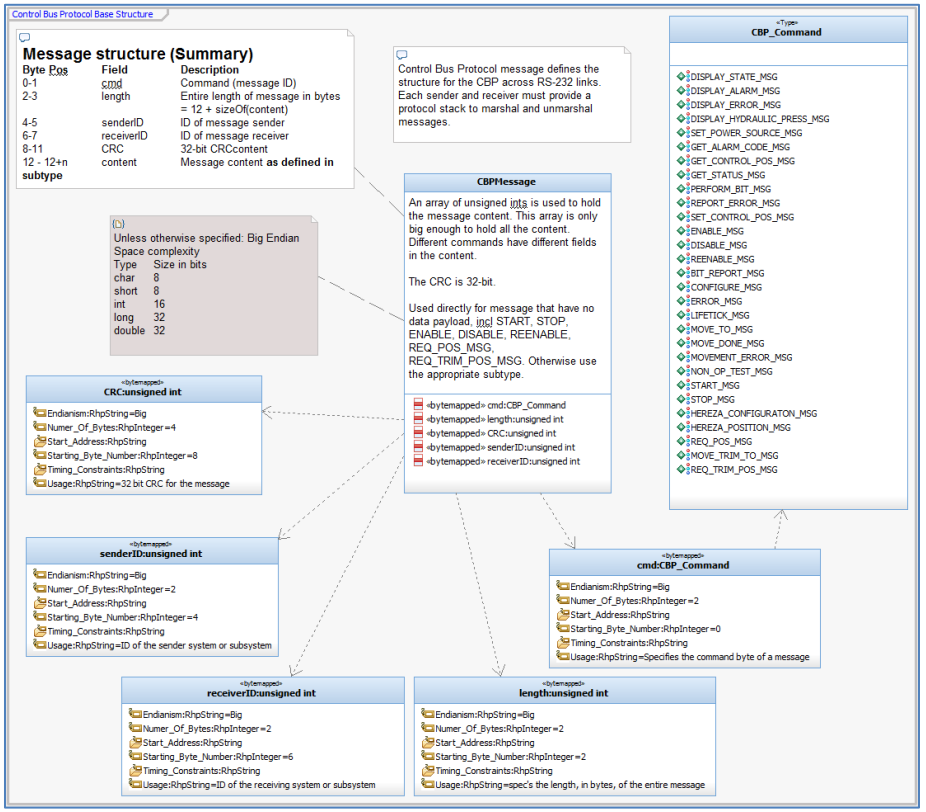


Figure 227: Base Structure of CBP Messages

Figure 228 shows the set of CBP messages. All of the defined subtypes provide their own contents structure.

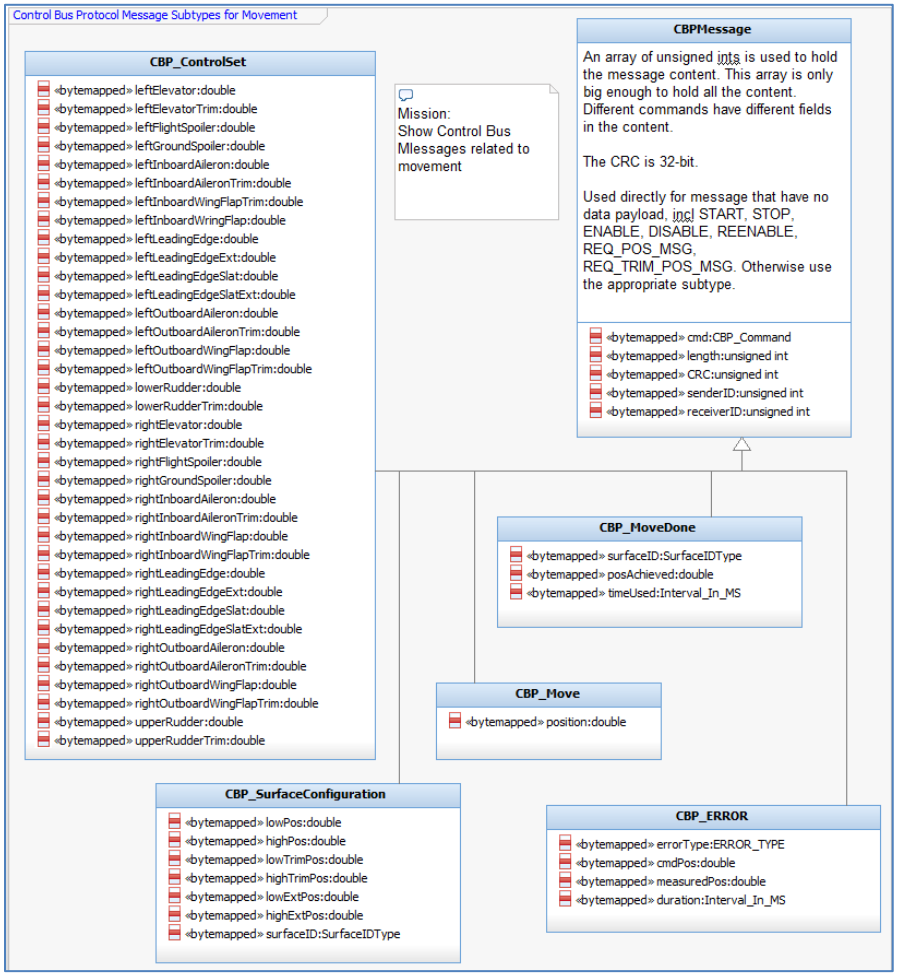


Figure 228: CBP Message Types related to movement

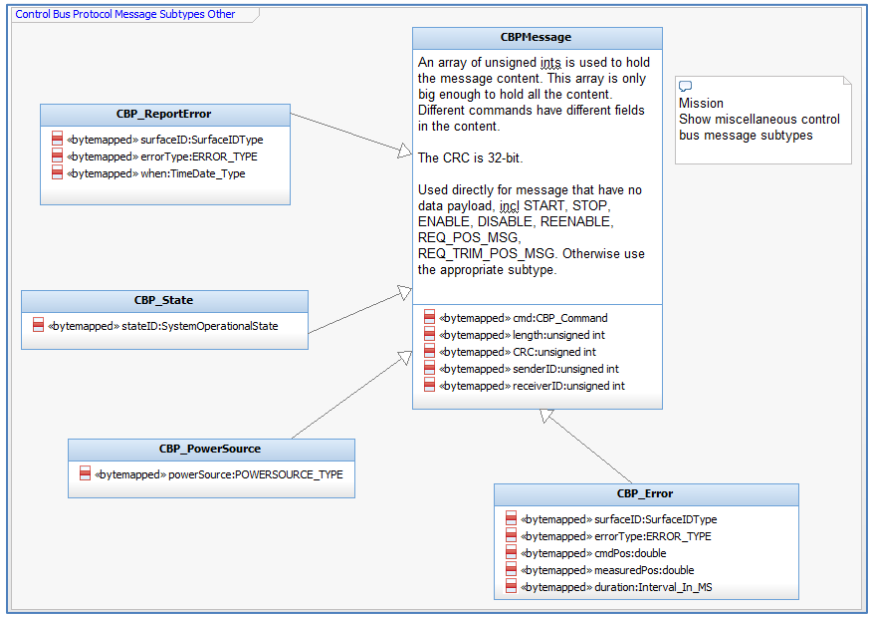
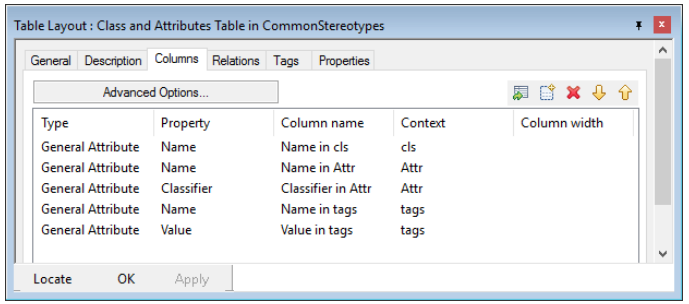


Figure 229: Other CBP Messages

It is also possible show this as a table with message type, list of attributes, type and description and other properties.

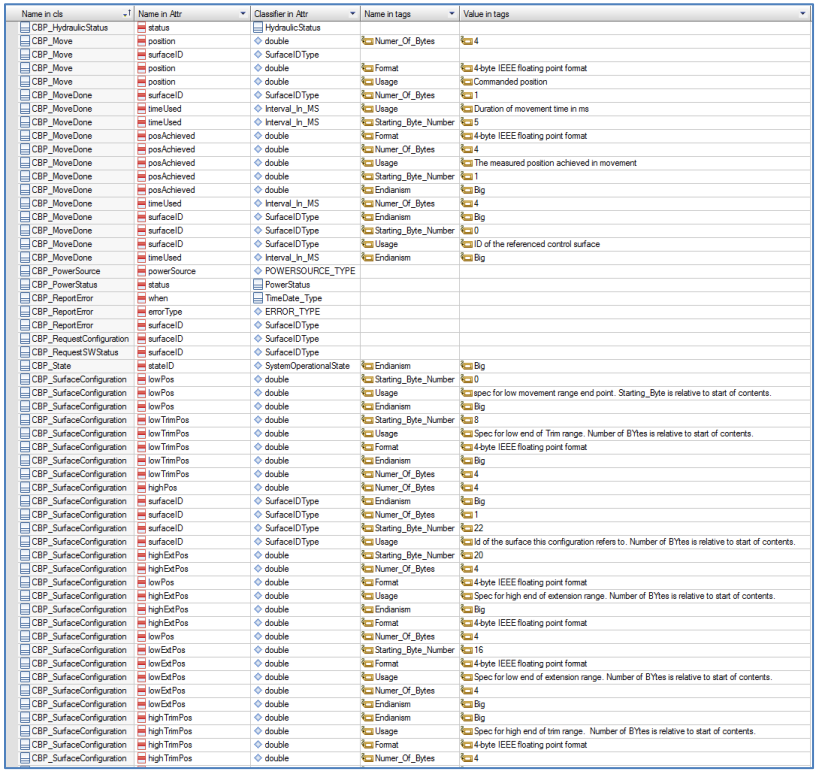
- In the **CommonStereotypes** package, add a new table layout named **Class and Attributes Table**.
- In the Columns Advanced Options set the following context pattern:
  - {pkg}Package*, {cls}Class, {Attr}Attribute*, {tags}Tag
- Define the columns in the *Columns* tab



- Create a table view in the **MessageTypesPkg** named **Message Attributes Table**.

- Right click on the table view and select *Features*.
- In the *Scope* property of the *General* tab, set the scope to be the **MessageTypesPkg**. Click on *OK* to close the *Features* dialog.
- Double click on the table view to open it.

You should see a table of the message types, their attributes, types, and filled out tagged values. A limited snapshot of this table is shown below:



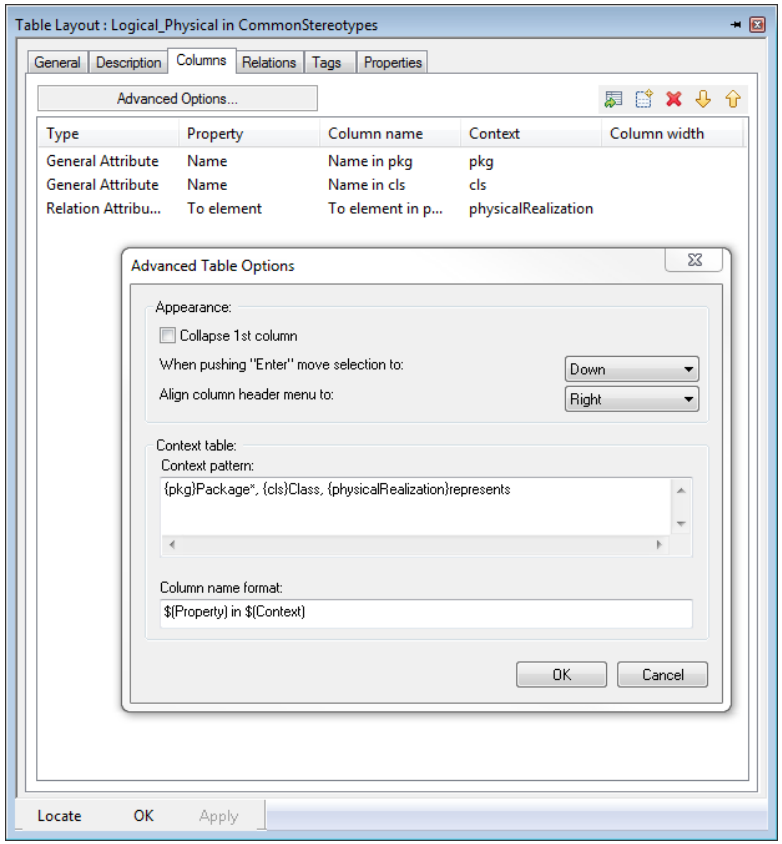
Name in cls	Name in Attr	Classifier in Attr	Name in tags	Value in tags
CBP_HydraulicStatus	status	HydraulicStatus		
CBP_Move	position	double	Numer_Of_Bytes	4
CBP_Move	surfaceID	SurfaceIDType		
CBP_Move	position	double	Format	4-byte IEEE floating point format
CBP_Move	position	double	Usage	Commanded position
CBP_MoveDone	surfaceID	SurfaceIDType	Numer_Of_Bytes	1
CBP_MoveDone	timeUsed	Interval_In_MS	Usage	Duration of movement time in ms
CBP_MoveDone	timeUsed	Interval_In_MS	Starting_Byte_Number	5
CBP_MoveDone	posAchieved	double	Format	4-byte IEEE floating point format
CBP_MoveDone	posAchieved	double	Numer_Of_Bytes	4
CBP_MoveDone	posAchieved	double	Usage	The measured position achieved in movement
CBP_MoveDone	posAchieved	double	Starting_Byte_Number	1
CBP_MoveDone	posAchieved	double	Endianism	Big
CBP_MoveDone	timeUsed	Interval_In_MS	Numer_Of_Bytes	4
CBP_MoveDone	surfaceID	SurfaceIDType	Endianism	Big
CBP_MoveDone	surfaceID	SurfaceIDType	Starting_Byte_Number	0
CBP_MoveDone	surfaceID	SurfaceIDType	Usage	ID of the referenced control surface
CBP_MoveDone	timeUsed	Interval_In_MS	Endianism	Big
CBP_PowerSource	powerSource	POWERSOURCE_TYPE		
CBP_PowerStatus	status	PowerStatus		
CBP_ReportError	when	TimeDate_Type		
CBP_ReportError	errorType	ERROR_TYPE		
CBP_ReportError	surfaceID	SurfaceIDType		
CBP_RequestConfiguration	surfaceID	SurfaceIDType		
CBP_RequestSWStatus	surfaceID	SurfaceIDType		
CBP_State	stateID	SystemOperationalState	Endianism	Big
CBP_SurfaceConfiguration	lowPos	double	Starting_Byte_Number	0
CBP_SurfaceConfiguration	lowPos	double	Usage	spec for low movement range end point. Starting_Byte is relative to start of contents.
CBP_SurfaceConfiguration	lowPos	double	Endianism	Big
CBP_SurfaceConfiguration	lowTrimPos	double	Starting_Byte_Number	8
CBP_SurfaceConfiguration	lowTrimPos	double	Usage	Spec for low end of Trim range. Number of BYtes is relative to start of contents.
CBP_SurfaceConfiguration	lowTrimPos	double	Format	4-byte IEEE floating point format
CBP_SurfaceConfiguration	lowTrimPos	double	Endianism	Big
CBP_SurfaceConfiguration	lowTrimPos	double	Numer_Of_Bytes	4
CBP_SurfaceConfiguration	highPos	double	Numer_Of_Bytes	4
CBP_SurfaceConfiguration	surfaceID	SurfaceIDType	Endianism	Big
CBP_SurfaceConfiguration	surfaceID	SurfaceIDType	Numer_Of_Bytes	1
CBP_SurfaceConfiguration	surfaceID	SurfaceIDType	Starting_Byte_Number	22
CBP_SurfaceConfiguration	surfaceID	SurfaceIDType	Usage	Id of the surface this configuration refers to. Number of BYtes is relative to start of contents.
CBP_SurfaceConfiguration	highExtPos	double	Starting_Byte_Number	20
CBP_SurfaceConfiguration	highExtPos	double	Numer_Of_Bytes	4
CBP_SurfaceConfiguration	lowPos	double	Format	4-byte IEEE floating point format
CBP_SurfaceConfiguration	highExtPos	double	Usage	Spec for high end of extension range. Number of BYtes is relative to start of contents.
CBP_SurfaceConfiguration	highExtPos	double	Endianism	Big
CBP_SurfaceConfiguration	highExtPos	double	Format	4-byte IEEE floating point format
CBP_SurfaceConfiguration	lowPos	double	Numer_Of_Bytes	4
CBP_SurfaceConfiguration	lowExtPos	double	Starting_Byte_Number	16
CBP_SurfaceConfiguration	lowExtPos	double	Format	4-byte IEEE floating point format
CBP_SurfaceConfiguration	lowExtPos	double	Usage	Spec for low end of extension range. Number of BYtes is relative to start of contents.
CBP_SurfaceConfiguration	lowExtPos	double	Numer_Of_Bytes	4
CBP_SurfaceConfiguration	lowExtPos	double	Endianism	Big
CBP_SurfaceConfiguration	highTrimPos	double	Endianism	Big
CBP_SurfaceConfiguration	highTrimPos	double	Usage	Spec for high end of trim range. Number of BYtes is relative to start of contents.
CBP_SurfaceConfiguration	highTrimPos	double	Format	4-byte IEEE floating point format
CBP_SurfaceConfiguration	highTrimPos	double	Numer_Of_Bytes	4

Figure 230: Message Attributes and Tags

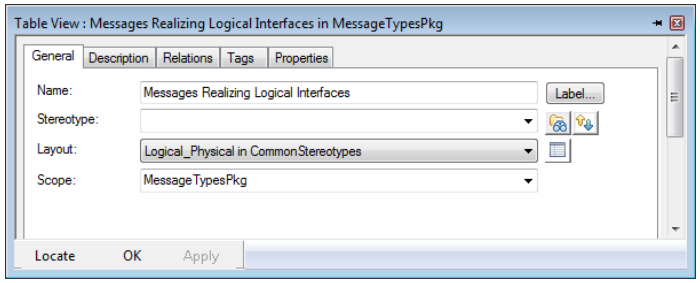
Other properties of interest can be easily added to the table, as desired.

It is important to show how these (physical) messages related to the (logical) services identified in the **InterfacesPkg**, referenced from the systems engineering model.

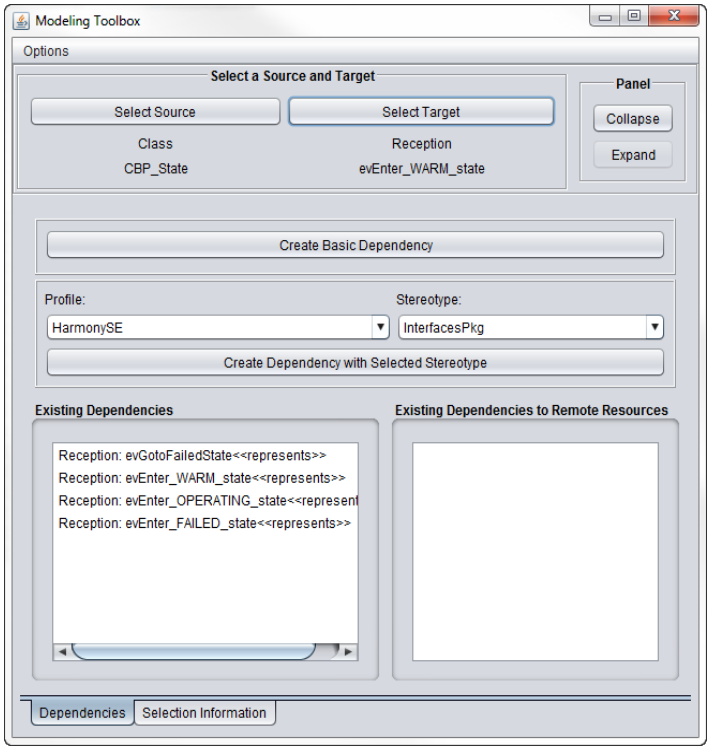
- Use the *File > Add Profile to Model* menu item to add the HarmonySE profile to the model (yes, it's ok to add this to a UML project as well. We want to use some of its features).
- In the **CommonStereotypesPkg**, add a new table layout named **Logical_Physical**



- Add a table view in the **MessageTypesPkg** using the above layout.



- Looking at the interface blocks (they're now classes) in the **IntefacesPkg** package, walk through all the event receptions, one by one and with the *Harmony Dependency* wizard, add a **«represents»** relation from one of the CBP messages to the event reception.
  - Ones that do not require a content payload can directly use the base class **CBPMessage** since the command field will identify which command is intended. All other messages will have to be subclassed from **CBPMessage** and have their additional contents defined.
  - If no **CBPMessage** subtype meets the need, then add a new one, defining its content fields and making it a subclass of **CBPMessage**



When complete, every event reception in every interface block will be represented by a **CBPMessage**:

Found 28 elements

Name in pkg	Name in PhysicalRepresentation	To element in LogicalRepresenta...
MessageTypesPkg		
	CBPMessage	checkPower
	CBPMessage	evRequestHydraulicStatus
	CBPMessage	reqENABLE_Command
	CBPMessage	request_Hydraulic_Status
	CBP_ControlSet	herezaPositionSet
	CBP_Error	evError
	CBP_HydraulicStatus	herezaHydraulic_Pressure
	CBP_HydraulicStatus	herezaHydraulic_Pressure
	CBP_Move	Command_To_Position
	CBP_Move	evMoveTo
	CBP_MoveDone	Updated_Position
	CBP_MoveDone	evMovementDone
	CBP_MoveDone	herezaPosition
	CBP_PowerSource	Select_Battery_As_Source
	CBP_PowerStatus	Update_Power_Status
	CBP_PowerStatus	updatePowerStatus
	CBP_ReportError	ReportError
	CBP_RequestConfiguration	Req_config_parameter
	CBP_RequestConfiguration	Req_config_parameters
	CBP_RequestSWStatus	request_SW_Integrity_Check
	CBP_RequestSWStatus	request_SW_Integrity_Check
	CBP_SWStatus	SW_Status
	CBP_SWStatus	SW_Status
	CBP_State	evEnter_FAILED_state
	CBP_State	evEnter_OPERATING_state
	CBP_State	evEnter_WARM_state
	CBP_State	evGotoFailedState
	CBP_SurfaceConfiguration	herezaConfiguration

Figure 231: Logical - Physical Schema Mapping Table

10.2.1.2 Power and Hydraulics Interfaces

Most, but not all services are modeled using event receptions in the logical interfaces. There are some, however, that are actual flows that are modeled as flow properties (attributes in UML). In this model, these are

- **iACES_Power_Aircraft_Power::power**
- **iACES_Hydraulics_Aircraft_Hydraulics::pressure**
- **iACES_ControlSurface_ACES_Power::power**
- **iACES_ControlSurface_ACES_Hydraulics::pressure**

The last two were added manually in the last chapter and you must move them from the **ACES_DecompositionPkg** to the **InterfacesPkg** to see them in the Shared model.

To provide those specifications is straightforward (far easier than the definiiton of the CBP messages we just performed.

In the **PhysicalInterfacesPkg** add the following Object Model Diagram (OMD) to create the classes

- **Hydraulic_Interface_Spec**
- **External_PowerInterface_Spec**
- **External_Power_Set**
    - Note that this contains a **frequency** attribute which is of type **Hertz**; this must be added to the types in the **PhysicalInterfaces** package. **Ampere** and **Pascal** are available from the SysML profile if you decide to add it to the model.
- Drag the appropriate interface blocks from the **InterfacesPkg** and add **«represents»** relations to those elements

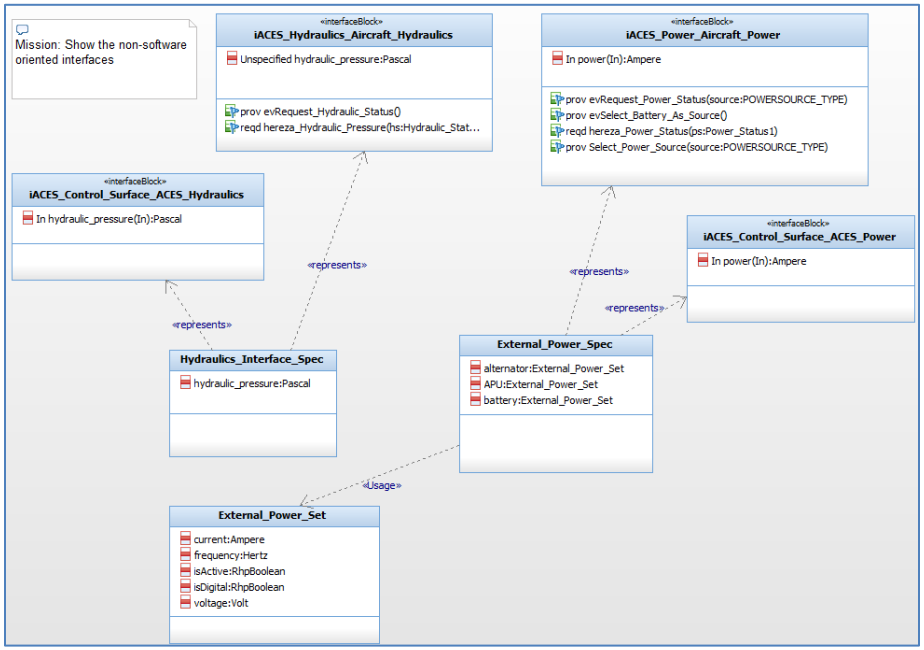


Figure 232: Hydraulic and Power Interfaces

Note that the stereotype InterfaceBlock is marked as Undefined. This is because we've not added the SysML profile to the model. This is not a problem but you can add the SysML profile using the *File > Add Profile to Model* feature of Rhapsody to resolve it. .

10.2.2 Specify the Physical Data Schema

We defined some basic types for a number of the attributes in the messages. Some are new, such as the **CBP_Command** which enumerates the different message type command fields, and others, such as **SurfaceID**, that are *copied* directly (and renamed slightly) from the referenced **InterfacesPkg > DataTypesPkg**.

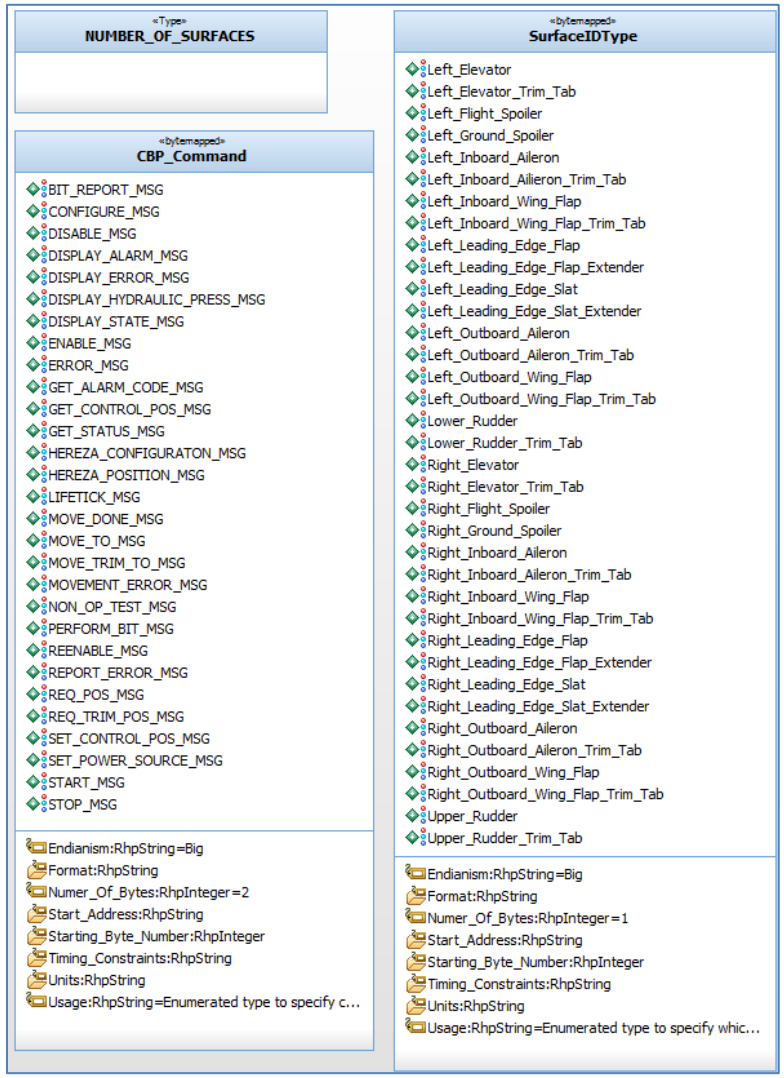


Figure 233: Some base types to support the physical data schema

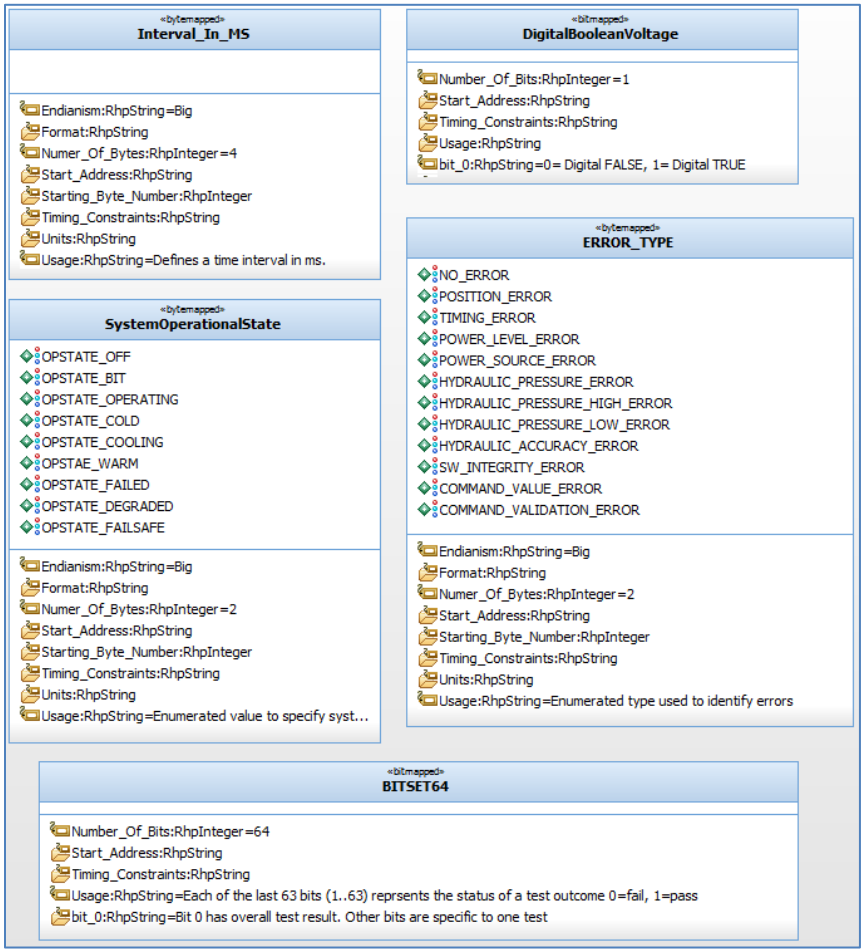


Figure 234: Some additional base types

10.3 Create the Subsystem Model

We created the subsystem models at the same time as the **Shared** model earlier in this chapter by using the SE Toolkit automation. If you have been doing things manually and have not yet created the subsystem models, now is the time.

There are a number of subsystem models to elaborate but we will do only one in this Deskbook – the **Control Surface Subsystem**. This was selected because it has an interesting deployment architecture with mechanical, electronic, and software aspects. Feel free to create the other subsystems when you've completed this Deskbook.

The structure of the subsystem models is all the same. As a starting point, the model has

- A copy (not a reference) of the subsystem specification package from the systems engineering model
- A reference to the **RequirementsAnalysisPkg** of the system engineering model (so the susbsytem can see its requirements)
- A reference to the **PhysicalInterfacesPkg** package of the Shared model
- A reference to the **CommonStereotypes** package of the Shared model
- An (empty) **SubsystemSpecPkg** to hold any additional requirements work that must be done
- An (empty) **DeploymentPkg** to hold the deployment architecture.

I like to enable browser ordering (*View > Browser Options > Enable Ordering*) to arrange the packages in this fashion:

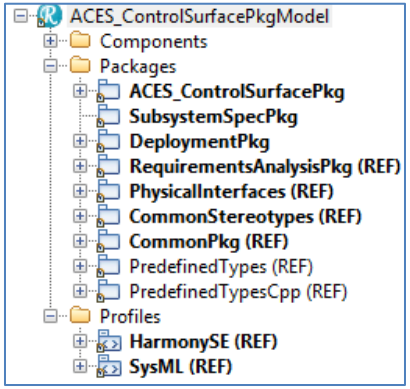
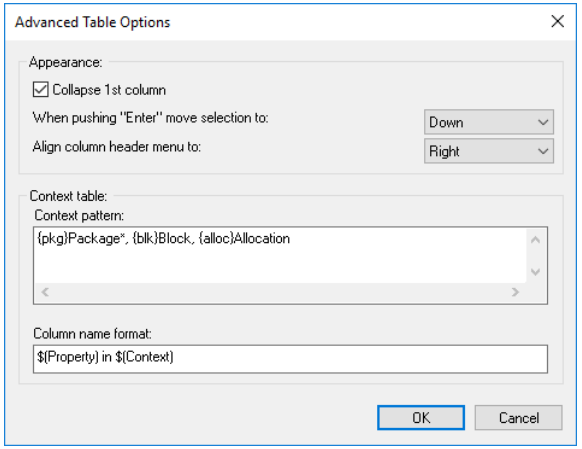


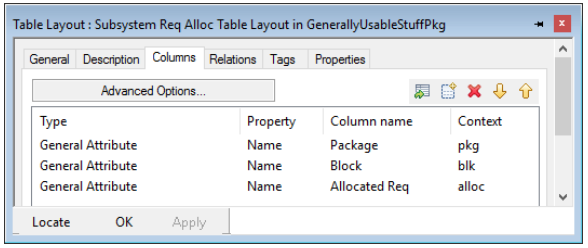
Figure 235: Subsystem Model Organization

I like to have a view of the requirements allocated to the subsystem. Such a table was defined in the *system* model **CommonPkg::Subsystem Req Alloc Table Layout.** I can include a reference to that. Complicating things just a bit, the relation between the subsystem and the requirement is **«allocate»**, which is defined in the SysML profile. So if you want to add this view, you'll need to add to model (by reference) the **CommonPkg::Subsystem Req Alloc Table Layout** (or duplicate it) and add SysML using *Add Profile to Model*.

To recap, the table layout was defined using the following context pattern:



and the following columns:



If you do that, you can create a table view that shows the requirements allocated to the subsystem:

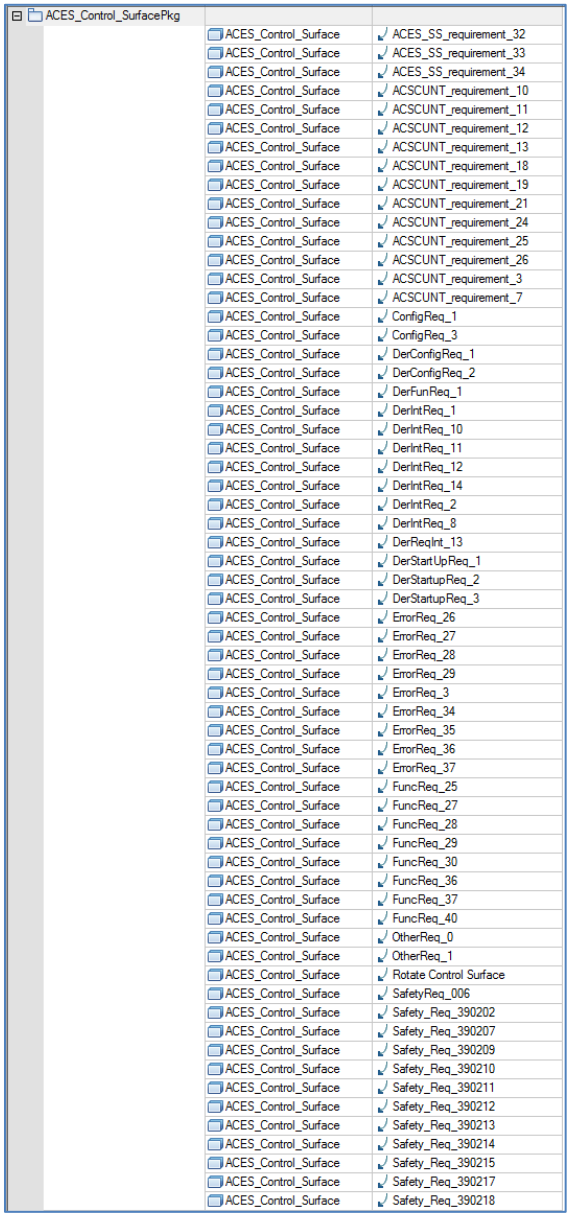
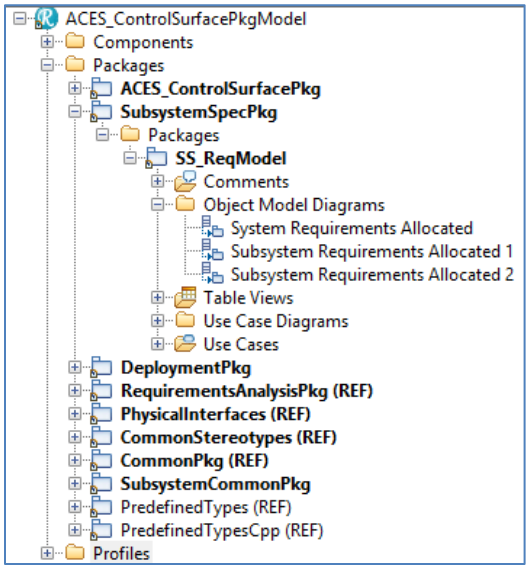


Figure 236: Requirements allocated to the Control Surface Subsystem

It is recommended that you place this table view in a package nested within in the **SubsystemSpecPkg** package, rather than **ACES_ControlSurfacePkg** because the latter is likely to be replaced in subsequent interations of the Harmony aMBSE process.

- Add a package named **SS_ReqModel** inside the SubsystemSpecPkg
- Place the above created table in that nested package



Another option for visualization of the requirements is to create some OMDs and add the requirements on to them. The next three figures show how this might be done.

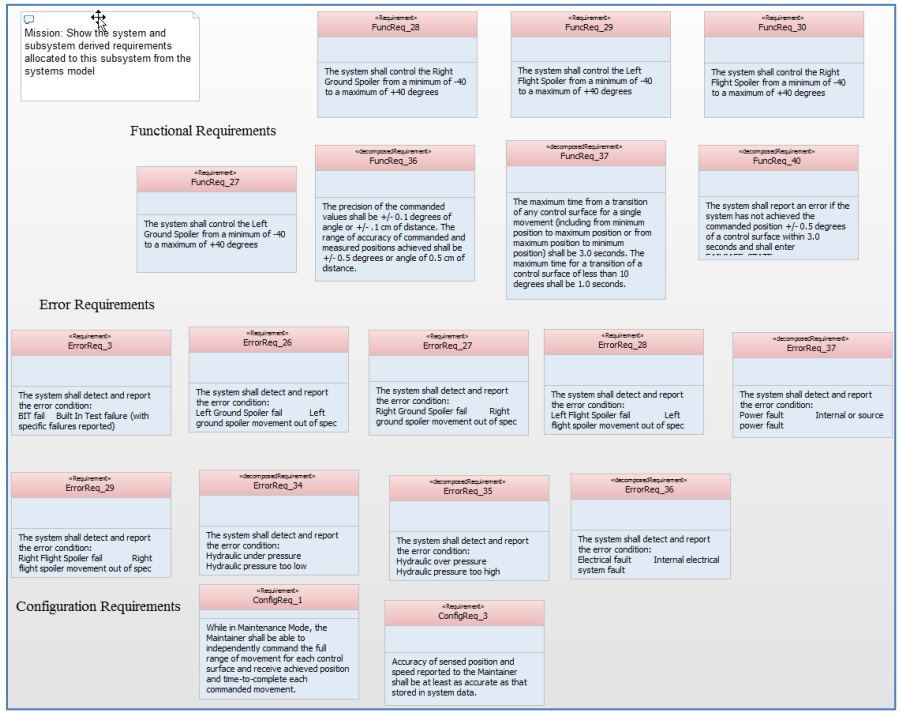


Figure 237: Subsystem requirements – 1

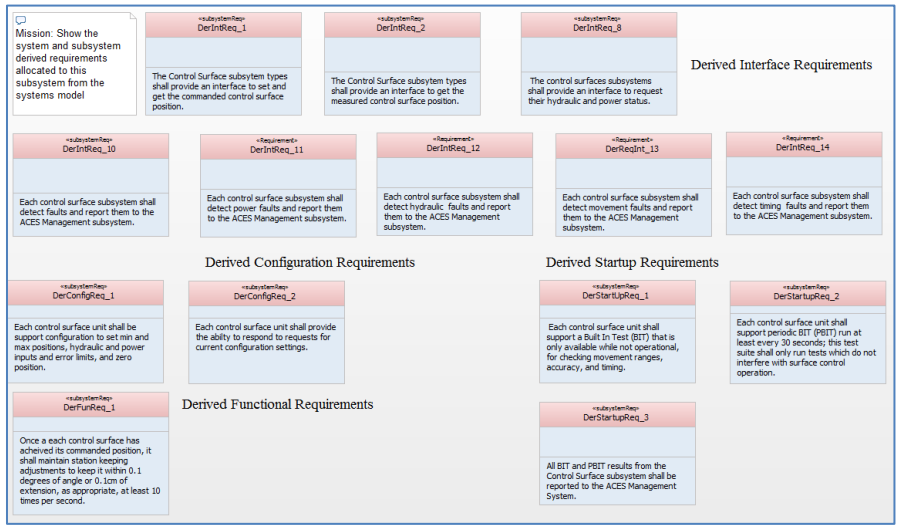


Figure 238: Subsystem Reqauirements - 2

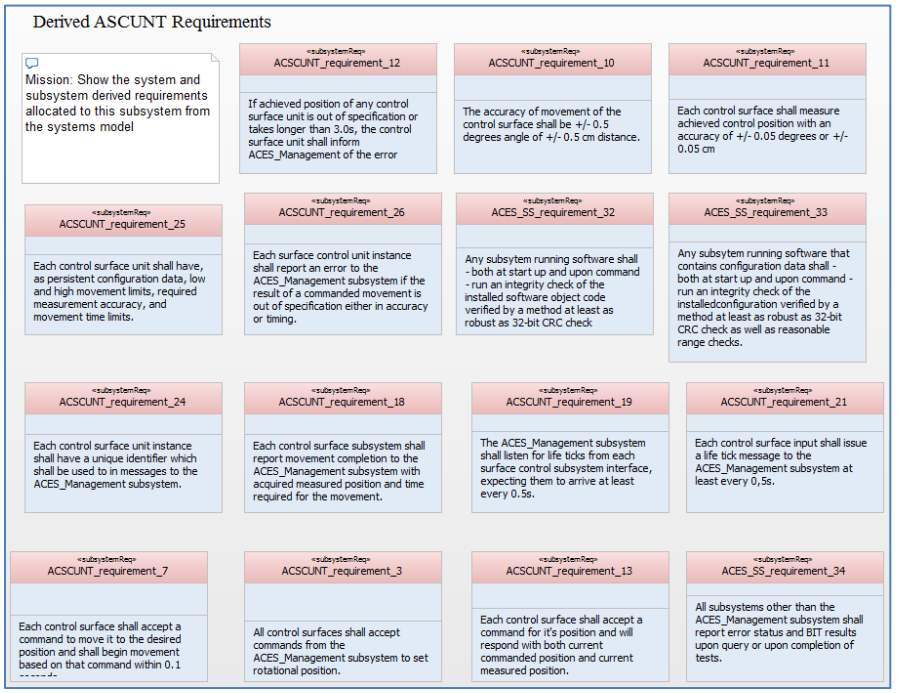


Figure 239: Subsystem Requirements - 3

If you draw these diagrams, they should also be placed in the **SubsystemSpecPkg::SS_ReqModel** package.

10.4 Define the Interdisciplinary Interfaces

This and the next section deal with the subsystem *deployment architecture*. The deployment architecture involves the

- Identification of the design work products (which we'll call *components* here) of different engineering disciplines
- The definition of the interdisciplinary interfaces between these components
- The allocation of requirements to the different components

What we do ***not*** want to do is to define the structure of the software, mechanical, electronic, hydraulic or pneumatic aspects; we have engineering specialists to do that after the hand off is complete. We want to specify these component just enough that we can do a good job of the tasks listed above. Specifically, this means that we will not define the internal software, mechanical or electronic structure here. That is important. Leave that works for the experts in those disciplines.

- In the **DeploymentPkg**, add a new Object Model Diagram (OMD) named **Deployment Architecture**.
- Fill out the diagram as shown in Figure 240

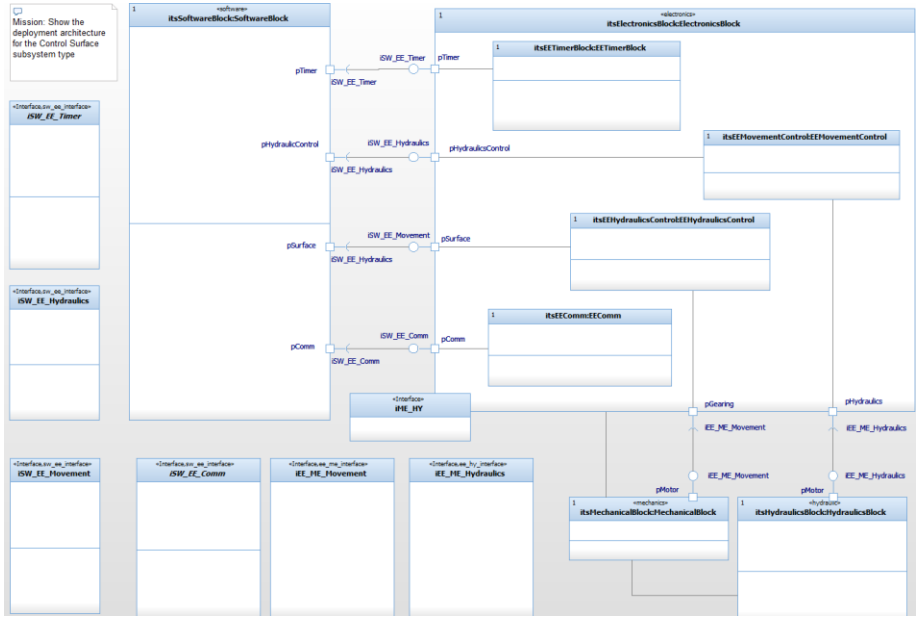


Figure 240: Deployment Architecture for the Control Surface Subsystem

Note that four discipline related components (shown as classes) are depicted in Figure 240:

- **Software Block**
- **Electronics Block**
- **Mechanical Block**

- **Hydraulics Block**

Each of these elements has an abbreviation of the discipline in the name, but more importantly carries a stereotype indicating it's domain: **«software», «electronic», «mechanics»,** and **«hydraulic».** Each of these is a stand-in for the collection of all design elements of the respective engineering discipline.

The **Electronics** block is broken down into 4 primary functional components:

- **Timer Block**
- **Movement Control Block**
- **Hydraulics Control Block**
- **Communications Block**

These should ***not*** be interpreted as a constraint on the electronic design. The only reason these functional components are identified at all is because the software-electronic interface is reasonably complex and identifiying these different components allows us to separate out the interfaces. This could have been done with a single port (one on the software block and one on the electronics block) and multiple provided and required interfaces. This could even have been done without creating the internal electronics components, but this does show the expected relation between the software commands intended to affect the mechanical and hydraulic parts (mediated by the electronics).

Note the direct associations between the electronic and mechanical blocks and the mechanical and hydraulic blocks. By convention (for deployment architecture only), I use ports to indicate connections that carry dynamic flows, such as software commands or mechanical force, and use direct associations to indicate non-behavioral (static) connections. Examples of static connections include cable management and mechanical fastenings. They are an important aspect of the design and so are represented on the deployment architecture.

Also shown are the currently empty interfaces. These interfaces must be detailed as to the information they carry and they means by which they are accessed by the participating disciplines.

10.4.1 Specifying the interfaces

It is important that the specification of interfaces – even when ultimately performed by the systems engineers – is done with the cooperation and agreement of engineers representing the affected disciplines. It is our experience that defining the interfaces late in the development process has been a leading cause of integration failure. Therefore, we will endeavor to do a good job of specifying the interfaces to clarify the anticipated collaboration of the designs from the contributing engineering disciplines.

Having said that, we also anticipate that the interfaces are likely to change. This is especially true in an incremental, iterative process. The keys to interdisplinary success are to

1. Specify the interfaces, including the services and physical implementation mechanisms using the best information currently available
2. Hide the actual designs behind the interfaces
3. Freeze the interfaces under configuration control
4. Later – if and when an issue with the interface is discovered – then thaw the interface from configuration control, renegotiate and refreeze the interface

Key Interface concept

The engineers on both sides of an interface should always have a known target to meet. It's ok if this target changes downstream in a controlled fashion with the knowledge and agreement of the affected parties.

To start with add a new package named **InterdisciplinaryInterfacesPkg** inside the **DeploymentPkg** package. Move all created interfaces there. This structuring will make it easier to create specific table views of the interface details later.

The software-electronic interfaces

The deployment architecture in Figure 240 identifies four separate sw-ee interfaces. In this section, we will detail those interfaces. To do so, we will use the stereotypes in the **CommonStereotypes** package such as **«memorymapped»** and **«interruptmapped»**. These define the interface metadata of interest.

First, let's look at the timer services, as specified in the interface **iSE_EE_Timer**. This is shown in Figure 241.

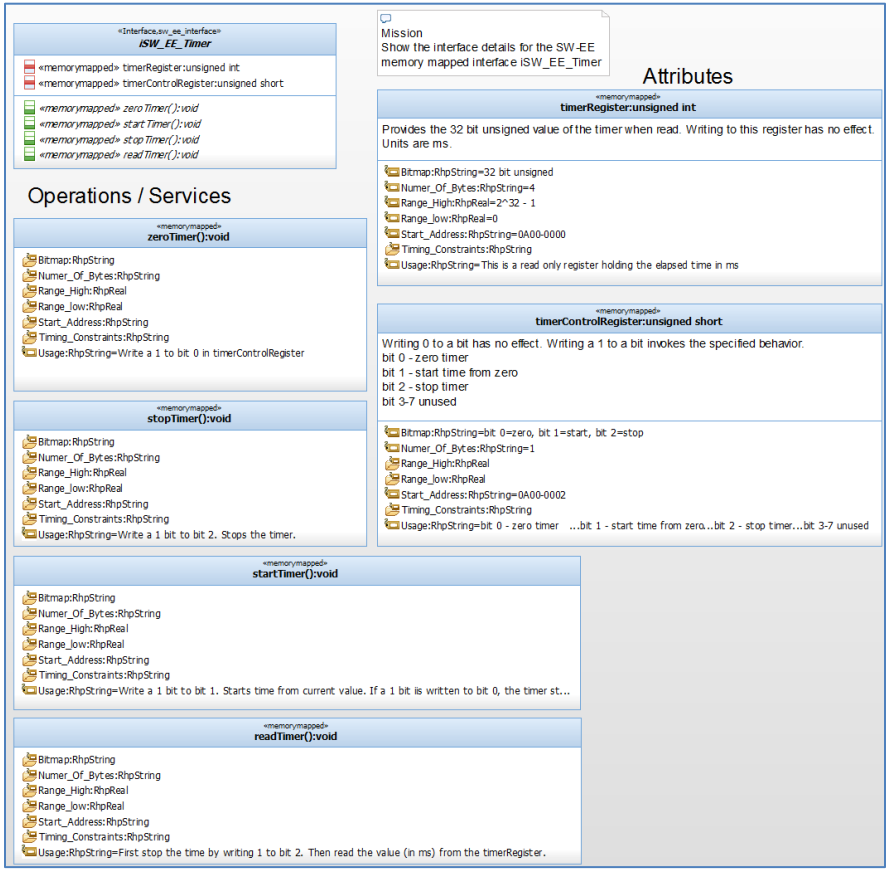
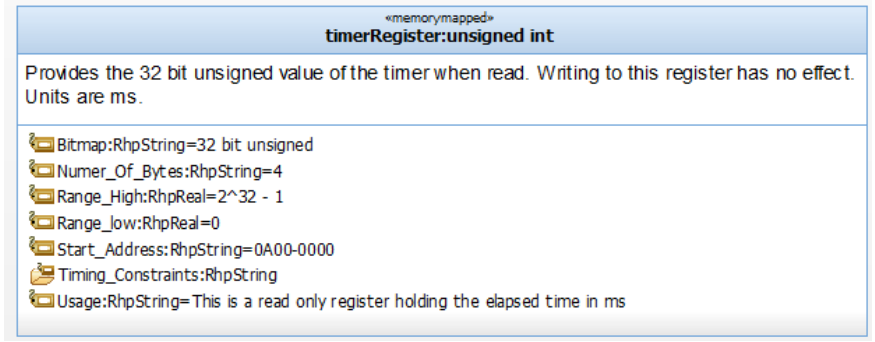


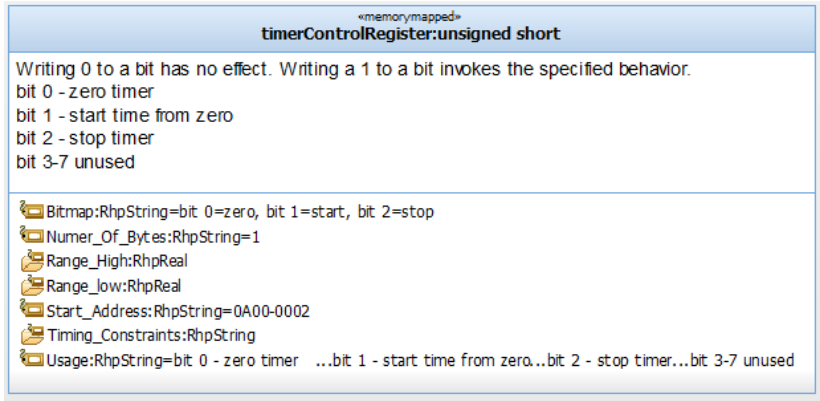
Figure 241: Details of iSW_EE_Timer interface

The tags defined by the **«memorymapped»** stereotype define the interface. There are two memory mapped attribute. The first is **timerRegister**:



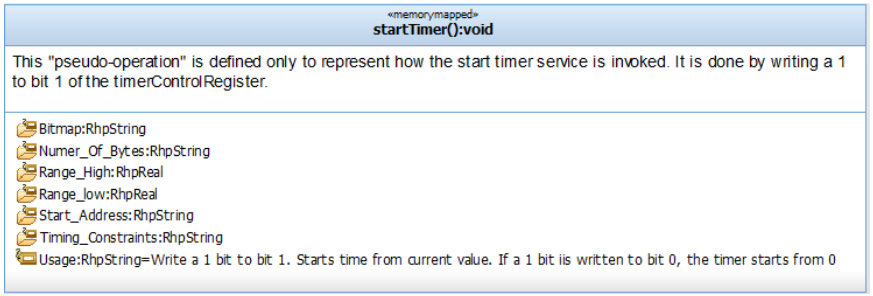
We see that this electronic register is 32 bits wide, at address 0A00:0000 and holds the current timer value when read.

The other attribute is the **timerControlRegister**. It is 8 bits wide and is located at address 0A00:0002. It has the following bits: writing a 1 bit to bit 0 sets the timer value to 0; writing a 1 bit to bit 1 starts the timer; writing a 1 to bit 2 stops the timer. Writing a zero value to a bit has no effect.



The operations defined merely use the memory mapped registers. They are not invoked as normal operations, but they specify how to invoke services.

For example to start the timer with the **startTimer** operation, it is really meant that the software will write a 1 to bit 1 at address 0A00-0002.



Next, let's look at the **iSW_EE_Comm** interface. This allows the software to send messages out through the communications bus. There are four memory-mapped attribues:

- **controlRegister** – sets the properties of the communications, including
  - Parity (on/off, and even/odd),
  - LSB/MSB first,
  - Data Length (7 or 8 bits)
  - Channel selection (0-15)
- **statusRegister** – returns the status of the communications
  - Loopback (on/off)
  - Framing error
  - Overrun error
  - Parity error
- **receiveBuffer** – where values appear when received
- **transmitBuffer** – wher values are written to be sent

Figure 242 shows the details, mostly stored in the tags from the relevant stereotypes. The operations are detailed but not shown, since they primarily just access the attributes. Note, however, that the operation **incomingValueReady() i**s interrupt-mapped. When interrupts are enabled in the control register and a value is received, the specified interrupt is invoked.

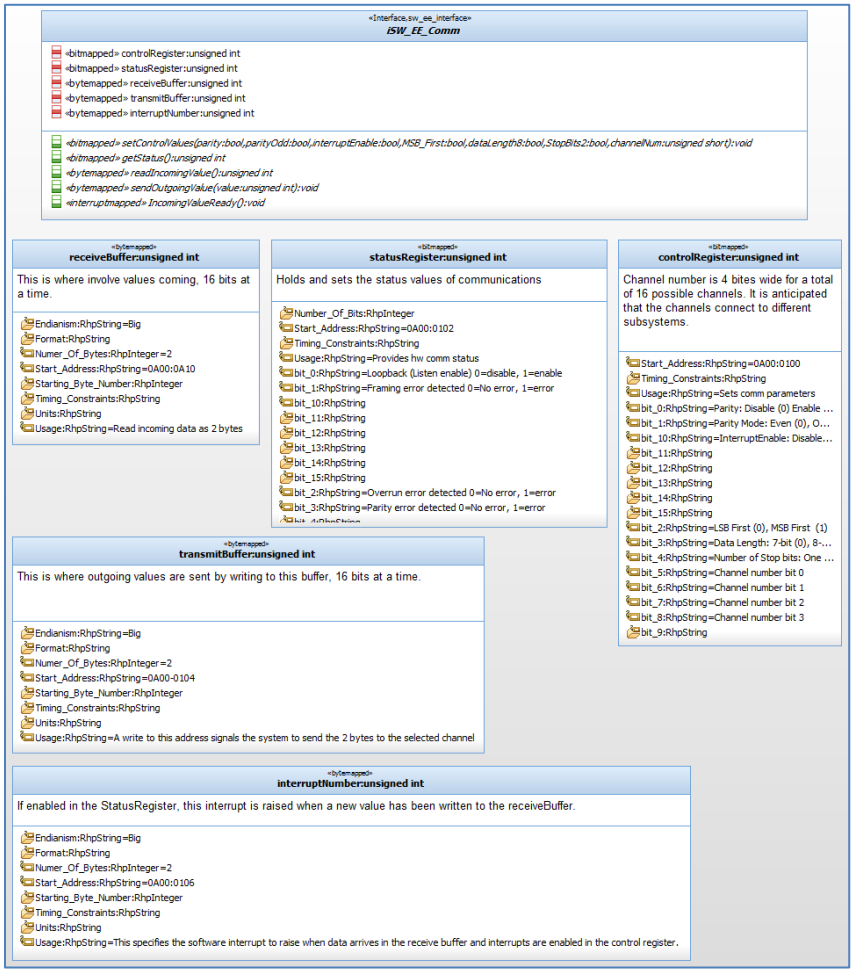


Figure 242: Details of iSW_EE_Comm interface

The **iSW_EE_Hydraulics** and **iSW_EE_Movement** interfaces are a bit simpler:

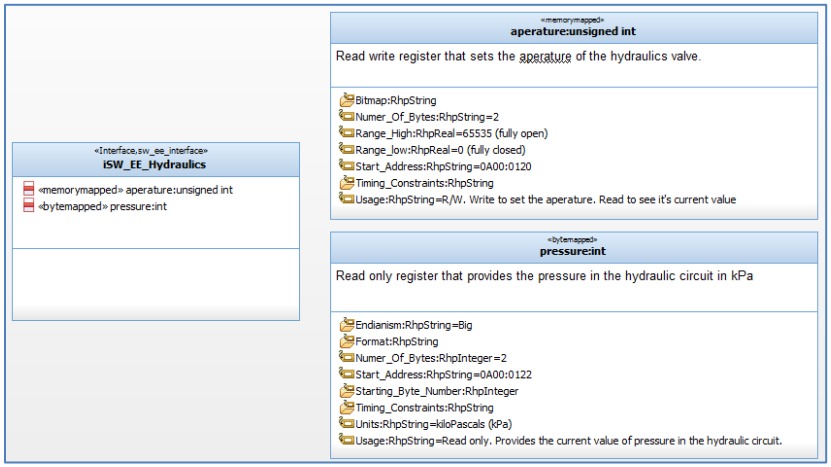


Figure 243: Details of iSW_EE_Hydraulics interface

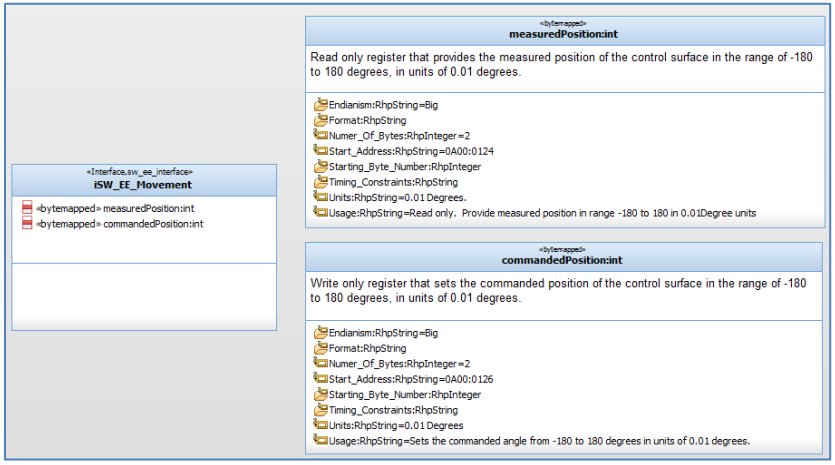
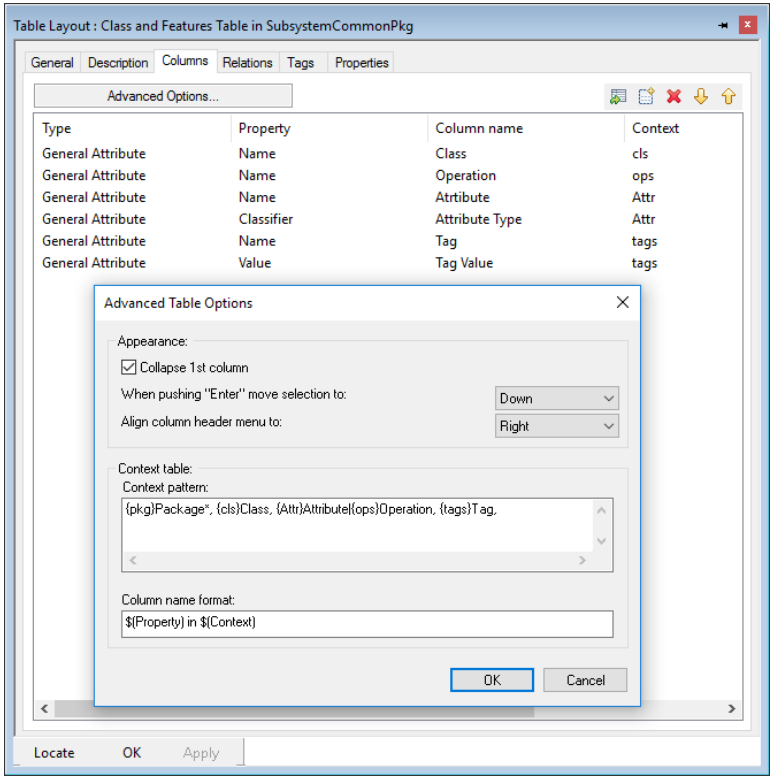


Figure 244: Details of iSW_EE_Movement interface

Showing the interfaces in a table

These interfaces can be shown in a table format as well as diagrammatically. This will be similar to the **Class and Attributes table layout** we used in the Shared model, but in this one we want to show operations as well.

- Add a new top level package named **SubsystemCommonPkg**.
- Right click on the new package and select *Add New > Tables and Charts > Table Layout.* Name this layout **Class And Features Layout**.
- Set the context pattern and columns and shown below



Especially note the use of the "|" (vertical bar) as an "or" operator in the context pattern.

- Add a new table view in the **DeploymentPkg > InterdisciplinaryInterfacesPkg** named **Interdisciplinary Interface Details**.
- Right click on the new table view and set the scope to be the **InterdisciplinaryInterfacesPkg** package and the layout to be **Class And Features Layout**.

The table, shown in the next two figures, should look something like this:

Class	Operation	Attribute	Attribute Type	Tag	Tag Value
iEE_ME_Hydraulics					
iEE_ME_Movement					
		rotationRate	int		
		valveAperature	double	High_Range	2.5
		valveAperature	double	Low_Range	0
		valveAperature	double	Units	cm
		valveAperature	double	accuracy	0.01
		valveAperature	double	precision	0.01
iSW_EE_Comm					
		controlRegister	unsigned int	Number_Of_Bits	5
		controlRegister	unsigned int	Start_Address	0A00:0100
		controlRegister	unsigned int	Usage	Sets comm parameters
		controlRegister	unsigned int	bit_0	Parity: Disable (0) Enable (1)
		controlRegister	unsigned int	bit_1	Parity Mode: Even (0), Odd (1)
		controlRegister	unsigned int	bit_10	InterruptEnable: Disable (0), Enable (1)
		controlRegister	unsigned int	bit_2	LSB First (0), MSB First (1)
		controlRegister	unsigned int	bit_3	Data Length: 7-bit (0), 8-bit (1)
		controlRegister	unsigned int	bit_4	Number of Stop bits: One stop bit (0), two stop bits (1)
		controlRegister	unsigned int	bit_5	Channel number bit 0
		controlRegister	unsigned int	bit_6	Channel number bit 1
		controlRegister	unsigned int	bit_7	Channel number bit 2
		controlRegister	unsigned int	bit_8	Channel number bit 3
		interruptNumber	unsigned int	Numer_Of_Bytes	2
		interruptNumber	unsigned int	Start_Address	0A00:0106
		interruptNumber	unsigned int	Usage	This specifies the software interrupt to raise when data arrives in the receive buffer and interrupts are enabled in the
		receiveBuffer	unsigned int	Numer_Of_Bytes	2
		receiveBuffer	unsigned int	Start_Address	0A00:0A10
		statusRegister	unsigned int	Start_Address	0A00:0102
		statusRegister	unsigned int	Usage	Provides hw comm status
		statusRegister	unsigned int	bit_0	Loopback (Listen enable) 0=disable, 1=enable
		statusRegister	unsigned int	bit_1	Framing error detected 0=No error, 1=error
		statusRegister	unsigned int	bit_10	
		statusRegister	unsigned int	bit_2	Overrun error detected 0=No error, 1=error
		statusRegister	unsigned int	bit_3	Parity error detected 0=No error, 1=error
		transmitBuffer	unsigned int	Numer_Of_Bytes	2
		transmitBuffer	unsigned int	Start_Address	0A00-0104
		transmitBuffer	unsigned int	Usage	A write to this address signals the system to send the 2 bytes to the selected
	IncomingValueReady				
	getStatus				
	readIncomingValue				
	sendOutgoingValue				
	setControlValues	receiveBuffer	unsigned int	Usage	Read incoming data as 2 bytes
iSW_EE_Hydraulics					
		aperature	unsigned int	Numer_Of_Bytes	2
		aperature	unsigned int	Range_High	65535 (fully open)
		aperature	unsigned int	Range_low	0 (fully closed)
		aperature	unsigned int	Start_Address	0A00:0120
		aperature	unsigned int	Usage	R/W. Write to set the aperature. Read to see it's current value
		pressure	int	Numer_Of_Bytes	2
		pressure	int	Start_Address	0A00:0122
		pressure	int	Units	kiloPascals (kPa)
		pressure	int	Usage	Read only. Provides the current value of pressure in the hydraulic circuit.

Figure 245: Interdisciplinary Interfaces - Part 1

Class	Operation	Attribute	Attribute Type	Tag	Tag Value
iSW_EE_Movement					
		commandedPositi...	int	Numer_Of_Bytes	2
		commandedPositi...	int	Start_Address	0A00:0126
		commandedPositi...	int	Units	0.01 Degrees
		commandedPositi...	int	Usage	Sets the commanded angle from -180 to 180 degrees in units of 0.01 degrees.
		measuredPosition	int	Numer_Of_Bytes	2
		measuredPosition	int	Start_Address	0A00:0124
		measuredPosition	int	Units	0.01 Degrees.
		measuredPosition	int	Usage	Read only. Provide measured position in range -180 to 180 in 0.01Degree units
iSW_EE_Timer					
		timerControlRegis...	unsigned short	Bitmap	bit 0=zero, bit 1=start, bit 2=stop
		timerControlRegis...	unsigned short	Numer_Of_Bytes	1
		timerControlRegis...	unsigned short	Start_Address	0A00-0002
		timerControlRegis...	unsigned short	Usage	bit 0 - zero timer bit 1 - start time from zero bit 2 - stop timer bit 3-7 unused
		timerRegister	unsigned int	Bitmap	32 bit unsigned
		timerRegister	unsigned int	Numer_Of_Bytes	4
		timerRegister	unsigned int	Range_High	2^32 - 1
		timerRegister	unsigned int	Range_low	0
		timerRegister	unsigned int	Start_Address	0A00-0000
		timerRegister	unsigned int	Usage	This is a read only register holding the elapsed time in ms
	readTimer	timerRegister	unsigned int	Usage	First stop the time by writing 1 to bit 2. Then read the value (in ms) from the
	startTimer			Usage	Write a 1 bit to bit 1. Starts time from current value. If a 1 bit iis written to bit 0, the timer starts from 0
	stopTimer			Usage	Write a 1 bit to bit 2. Stops the timer.
	zeroTimer			Usage	Write a 1 to bit 0 in timerControlRegister

Figure 246: Interdisciplinary Interfaces - Part 2

10.5 Allocate Requirements to Engineering Disciplines

Each engineering discipline within the subsystem must also know their requirements. You must take each requirement allocated to the subsystem and either allocate it directly to an engineering discipline or decompose it into derived requirments that are then so allocated. In a SysML model, this can be done on a Requirements Diagram or in a table. In this UML subsystem model, it can be done on an OMD or in a table.

Personally, I prefer to create the new requirements on diagrams but later view the results in tabular form. This work should be done in the **SubsystemSpecPkg > SS_ReqModel** package. This package already contains the diagrams showing the allocated requirements and the **Subsystem Requirements Table View**, previously created. How I like to perform this task:

- Create a new diagram for the purpose of allocation to the engineering disciplines
- Drag the blocks representing the disciplines onto the diagram
- One at a time, drag a requirement onto the diagram and either
  - Create an allocate relationship from the block to the requirement, OR
  - Create new, derived requirements
  - Add a *derive* relation between the original and the new requirement(s)
  - Draw an allocate relation from the engineering discipline blocks to the new requirement(s)
- Add new diagrams as necessary
- Repeat until all requirements are allocated

The diagrams below are typical of this effort.

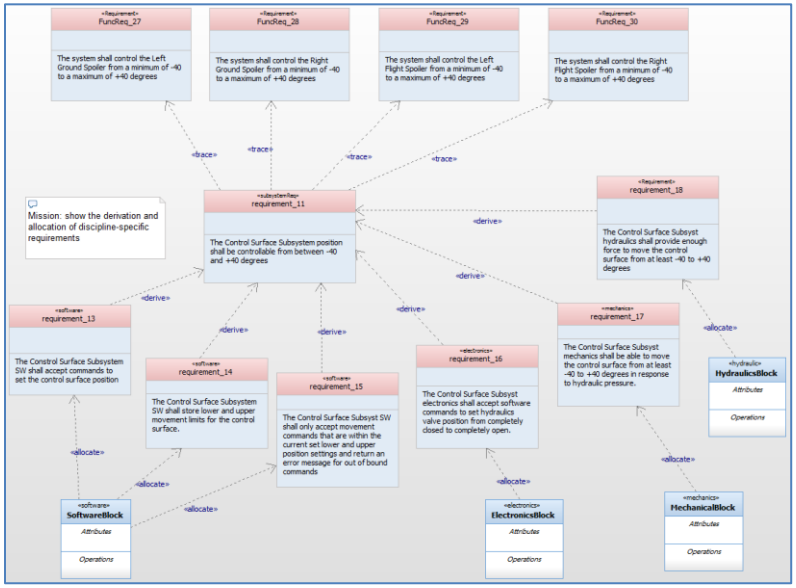


Figure 247: Derivation of Discipline-Specific Requirements - 1

Note that in Figure 247 requirement_11 is really an abstraction of all the specific requirements about the subsystem movement ranges, and then discipline-specific requirements are derived from that. Then those new requirements are allocated to the different engineering discipline blocks.

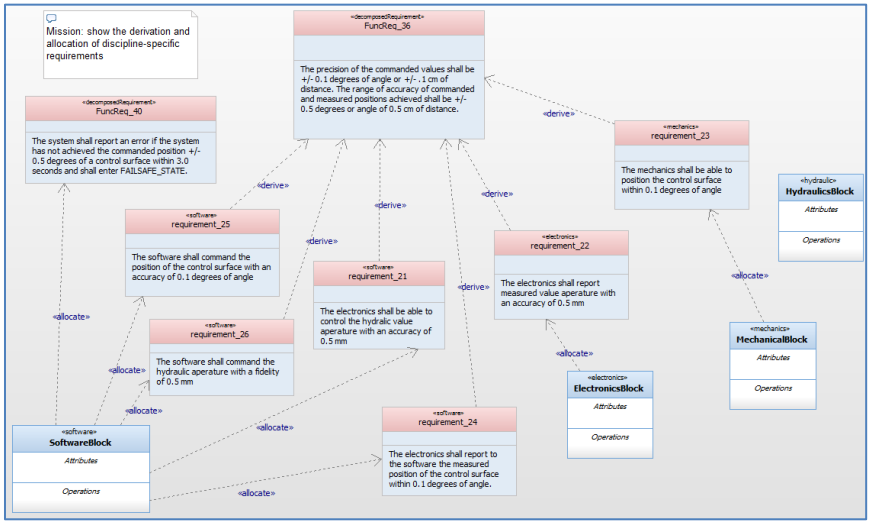


Figure 248: Derivation of Discipline-Specific Requirements - 2

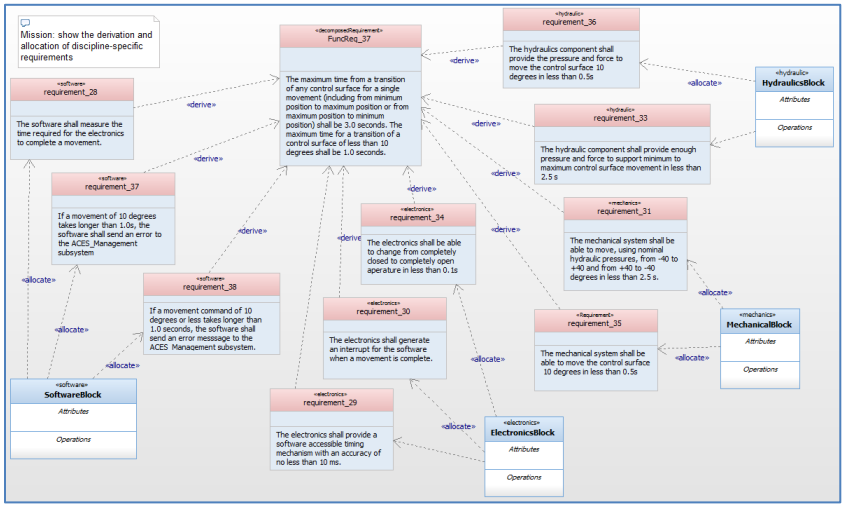


Figure 249: Derivation of Discipline-Specific Requirements - 3

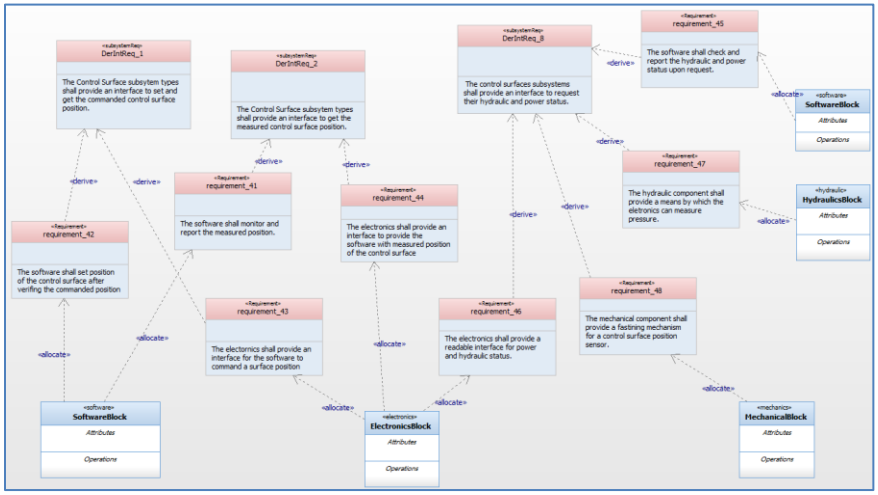


Figure 250: Derivation of Discipline-Specific Requirements - 4

There are, of course, more requirements to derive and allocate but the previous four slides shows the work that must be done.

Of course, this data can be visualized in tables. We've covered requirements tables previously, so we'll just show the allocation table.

- Right click on the **SubsystemSpecPkg > SS_ReqModel** package and select *Add New > Views and Layouts > Table View*
- Name this table view **Discipline Requirements Allocation Table**.
- Right click on the new table and select *Features*.
- In the *Features* dialog, set the *Table Layout* to the **Alloc Table Layout** (which is located in the SysML Profile package).
- Set the scope to the entire model (default)
- Click *OK*

The table shows the all allocations in the model in the first column. The second column is the source of the allocation (mostly, the blocks representing the different engineering disciplines). The third column is the target (the requirement) of that allocation relation. The next two figures show the table contents.

Note that there are more requirments to be allocated but this is enough to show the approach.

Name	From	To	Description
ACES_SS_requirement_32	ACES_ControlSurface	ACES_SS_requirement_32	
ACES_SS_requirement_33	ACES_ControlSurface	ACES_SS_requirement_33	
ACES_SS_requirement_34	ACES_ControlSurface	ACES_SS_requirement_34	
ACSCUNT_requirement_10	ACES_ControlSurface	ACSCUNT_requirement_10	
ACSCUNT_requirement_11	ACES_ControlSurface	ACSCUNT_requirement_11	
ACSCUNT_requirement_12	ACES_ControlSurface	ACSCUNT_requirement_12	
ACSCUNT_requirement_13	ACES_ControlSurface	ACSCUNT_requirement_13	
ACSCUNT_requirement_18	ACES_ControlSurface	ACSCUNT_requirement_18	
ACSCUNT_requirement_19	ACES_ControlSurface	ACSCUNT_requirement_19	
ACSCUNT_requirement_21	ACES_ControlSurface	ACSCUNT_requirement_21	
ACSCUNT_requirement_24	ACES_ControlSurface	ACSCUNT_requirement_24	
ACSCUNT_requirement_25	ACES_ControlSurface	ACSCUNT_requirement_25	
ACSCUNT_requirement_26	ACES_ControlSurface	ACSCUNT_requirement_26	
ACSCUNT_requirement_3	ACES_ControlSurface	ACSCUNT_requirement_3	
ACSCUNT_requirement_7	ACES_ControlSurface	ACSCUNT_requirement_7	
ConfigReq_1	ACES_ControlSurface	ConfigReq_1	
ConfigReq_3	ACES_ControlSurface	ConfigReq_3	
DerConfigReq_1	ACES_ControlSurface	DerConfigReq_1	
DerConfigReq_2	ACES_ControlSurface	DerConfigReq_2	
DerFunReq_1	ACES_ControlSurface	DerFunReq_1	
DerIntReq_1	ACES_ControlSurface	DerIntReq_1	
DerIntReq_10	ACES_ControlSurface	DerIntReq_10	
DerIntReq_11	ACES_ControlSurface	DerIntReq_11	
DerIntReq_12	ACES_ControlSurface	DerIntReq_12	
DerIntReq_14	ACES_ControlSurface	DerIntReq_14	
DerIntReq_2	ACES_ControlSurface	DerIntReq_2	
DerIntReq_8	ACES_ControlSurface	DerIntReq_8	
DerReqInt_13	ACES_ControlSurface	DerReqInt_13	
DerStartUpReq_1	ACES_ControlSurface	DerStartUpReq_1	
DerStartupReq_2	ACES_ControlSurface	DerStartupReq_2	
DerStartupReq_3	ACES_ControlSurface	DerStartupReq_3	
ErrorReq_26	ACES_ControlSurface	ErrorReq_26	
ErrorReq_27	ACES_ControlSurface	ErrorReq_27	
ErrorReq_28	ACES_ControlSurface	ErrorReq_28	
ErrorReq_29	ACES_ControlSurface	ErrorReq_29	
ErrorReq_3	ACES_ControlSurface	ErrorReq_3	
ErrorReq_34	ACES_ControlSurface	ErrorReq_34	
ErrorReq_35	ACES_ControlSurface	ErrorReq_35	
ErrorReq_36	ACES_ControlSurface	ErrorReq_36	
ErrorReq_37	ACES_ControlSurface	ErrorReq_37	

Figure 251: Table of requirements allocated to engineering disciplines - 1

Name	From	To	Description
FuncReq_25	ACES_ControlSurface	FuncReq_25	
FuncReq_27	ACES_ControlSurface	FuncReq_27	
FuncReq_28	ACES_ControlSurface	FuncReq_28	
FuncReq_29	ACES_ControlSurface	FuncReq_29	
FuncReq_30	ACES_ControlSurface	FuncReq_30	
FuncReq_36	ACES_ControlSurface	FuncReq_36	
FuncReq_37	ACES_ControlSurface	FuncReq_37	
FuncReq_40	ACES_ControlSurface	FuncReq_40	
FuncReq_40	SoftwareBlock	FuncReq_40	
OtherReq_0	ACES_ControlSurface	OtherReq_0	
OtherReq_1	ACES_ControlSurface	OtherReq_1	
requirement_13	SoftwareBlock	requirement_13	
requirement_14	SoftwareBlock	requirement_14	
requirement_15	SoftwareBlock	requirement_15	
requirement_16	ElectronicsBlock	requirement_16	
requirement_17	MechanicalBlock	requirement_17	
requirement_18	HydraulicsBlock	requirement_18	
requirement_21	SoftwareBlock	requirement_21	
requirement_22	ElectronicsBlock	requirement_22	
requirement_23	MechanicalBlock	requirement_23	
requirement_24	SoftwareBlock	requirement_24	
requirement_25	SoftwareBlock	requirement_25	
requirement_26	SoftwareBlock	requirement_26	
requirement_28	SoftwareBlock	requirement_28	
requirement_30	ElectronicsBlock	requirement_30	
requirement_31	MechanicalBlock	requirement_31	
requirement_34	ElectronicsBlock	requirement_34	
requirement_35	MechanicalBlock	requirement_35	
requirement_36	HydraulicsBlock	requirement_36	
requirement_37	SoftwareBlock	requirement_37	
requirement_38	SoftwareBlock	requirement_38	
requirement_41	SoftwareBlock	requirement_41	
requirement_42	SoftwareBlock	requirement_42	
requirement_43	ElectronicsBlock	requirement_43	
requirement_44	ElectronicsBlock	requirement_44	
requirement_45	SoftwareBlock	requirement_45	
requirement_46	ElectronicsBlock	requirement_46	
requirement_47	HydraulicsBlock	requirement_47	
requirement_48	MechanicalBlock	requirement_48	

Figure 252: Table of requirements allocated to engineering disciplines - 2

11 Post Log: Where we go from here

We have only traversed two use cases through the first set of system engineering workflows. We only detailed a single subsystem, of several involved in realizing those use cases, and even for those, we didn't do a complete allocation. Nevertheless, you can see how the workflow unfolds. In a real system development, we would continue with the allocations for the **ACES_ControlSystem** and we would detail the other subsystems as well. At that point, the hand off work flow would be complete, and each of the subsystem teams can begin work.

11.1 Downstream engineering begins

The subsystem teams are generally interdisciplinary; that is, they have members who specialize in different engineering disciplines, such as software, electronics, mechanics, and hydraulics. At this point, the following models exist to support the detailed design and implementation by the subsystem teams:

- Shared Model
  - Physical Interfaces
    - Physical Types
  - Common Stereotypes
- Subsystem Models, each of which has
  - Requirements specification with allocated requirements
  - Deployment architecture identifying the involved disciplines
    - Interfaces between the engineering disciplines
    - Requirements allocated to the engineering disciplines

This is the information required to perform the downstream engineering work, so that later the different subsystems can be integrated and verified and validated as a whole.

11.2 System Engineering Continues

In general, we believe the best systems engineering process is one that is both incremental and iterative. In this Deskbook, we've walked through what one such iteration might look like. However, there are a number of other use cases and associated requirements that must be detailed. This means that this workflow will be repeated, resulting in an increasingly complete and comprehensive system specification and model(s).

At the end of most (although not necessarily all) iterations, a hand off workflow is performed to update the subsystem teams with their elaborated requirements so they can incrementally add those features and properties to their subsystems.

12 Appendix: Passing Data Around in Rhapsody for C++

The most common language version of Rhapsody in systems engineering is C++. This impacts systems engineers because both UML and SysML use an *action language* to specify primitive actions, including the content of actions in activity diagrams, in the implementation of functions and options, and in the action lists in state machines. Also there is a generic action language provided by Rhapsody. However, by far, most people just using the underlying target implementation language as the action language, for a variety of good reasons.

One outcome of this is to require the systems engineer to understand enough of the underlying action language to create and manipulate data types. This appendix is meant to give a brief introduction to the data typing and parameter passing in Rhapsody for C++ and is not meant to be a comprehensive discussion of C++ data typing.

12.1 Simple and Complex Types

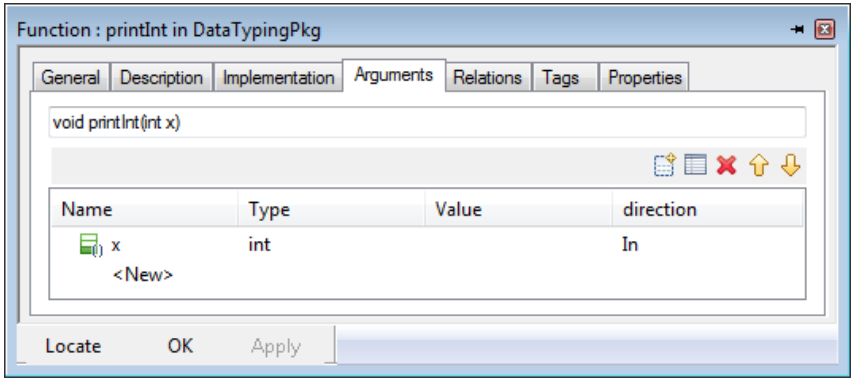
As far as Rhapsody is concerned, simple types are either ones directly providing by the underlying action language or map directly to them. Thus, simple types include:

- Language Independent Types
  - RhpAddress
  - RhpBoolean
  - RhpInteger
  - RhpPositive
  - RhpReal
  - RhpUnlimitedNatural
  - OMBoolean
- Language Dependent Types
  - bool
  - int
  - long
  - short
  - float
  - long double
  - short
  - unsigned char
  - unsigned int
  - unsigned long
  - unsigned short

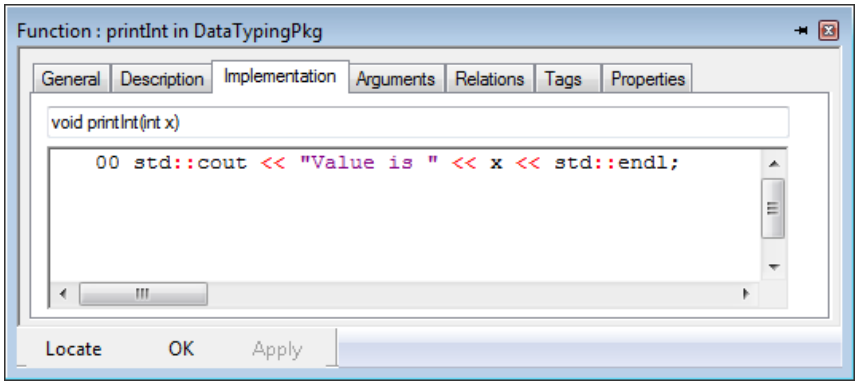
Complex Types are all other types. The reason why it matters is that when you add an argument to an operation or event, simple types get copied and sent, while for complex types, a reference to the original value is sent instead.

In (Input) Parameters

For example, if I create a function **printInt** that takes an argument of type **int** and prints it, I create the function like this:



I can use the passed value **x** in the implementation directly like this:



Here's the generated code for the function:

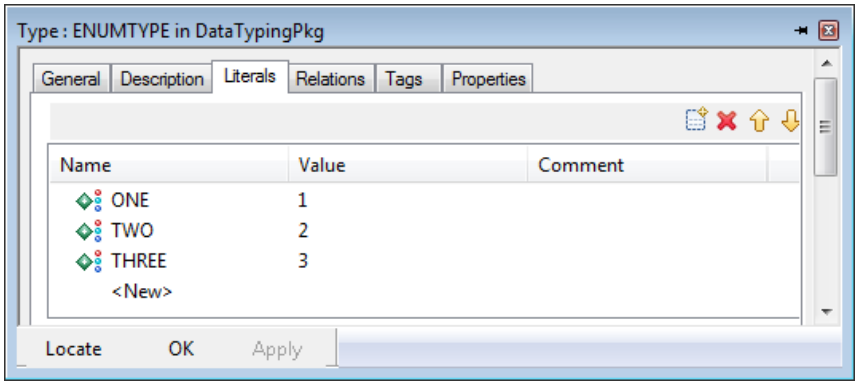
```
//## operation printInt(int)
void printInt(int x) {
    //#[ operation printInt(int)
    std::cout << "Value is " << x << std::endl;
    //#]
}
```

And I can invoke the function like this

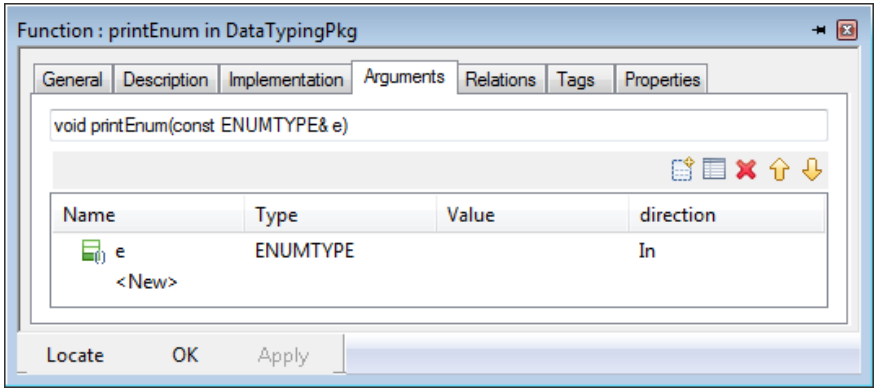
```
printInt(5);
```

However, if instead I use a complex type (in this case, I created an enumeration type call **ENUMTYPE** with values like "ONE", "TWO" and "THREE", things are different.

Here's the type definition:



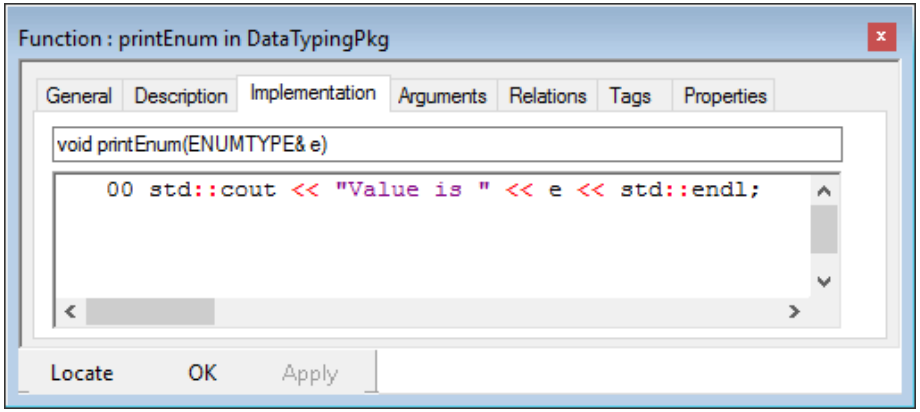
And the **printEnum** function parameter



With an implementation that looks like this.

```
void printEnum(const ENUMTYPE& e) {
    //#[ operation printEnum(ENUMTYPE)
    std::cout << "Value is " << e << std::endl;
    //#]
}
```

See that "&" symbol? That indicates that we are passed a *constant reference* to the value in **e**. We can treat **e** just like we did **x** in the previous example.

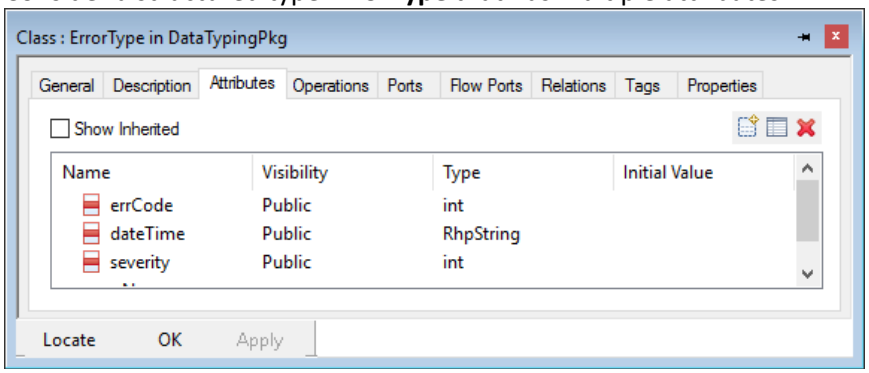


To call **printEnum**, I can just just

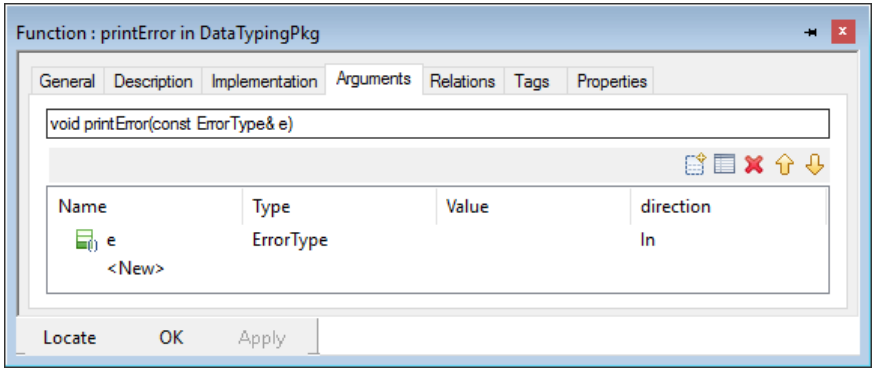
```
printEnum(myValue);
```

Let's now consider a structured value. In C++, a structured value is either a *struct* or a *class*. These two things are essentially the same (they only differ on the default visibility of the features; *struct* features are *public* by default while *class* features are *private* by default).

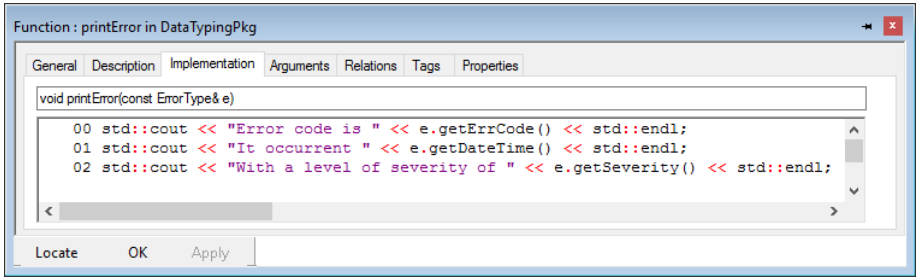
Consider a structured type **ErrorType** that has multiple attributes:



We can write a **printError** function that receives a single argument **e** of type **ErrorType**



and a simple implementation:

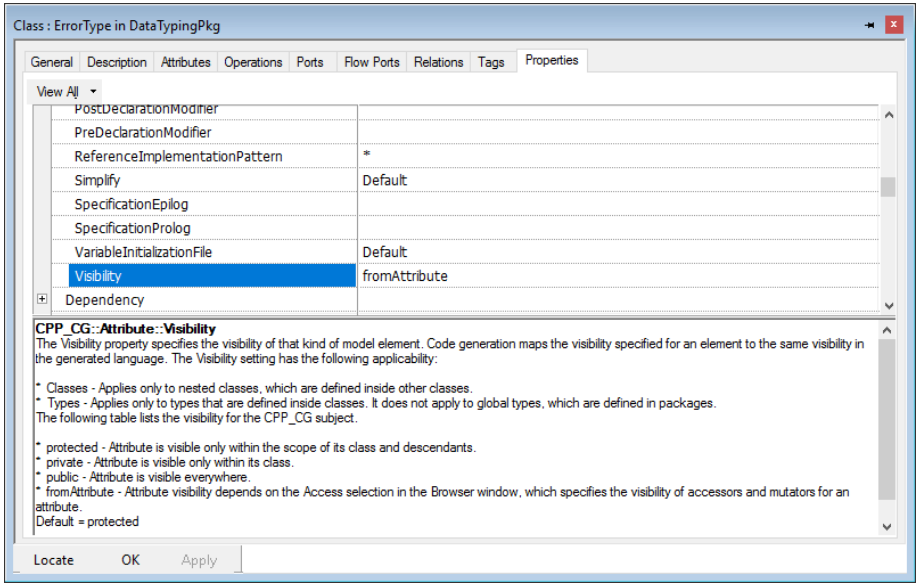


Why didn't we just directly access the values of **e.errCode**, **e.dateTime** and **e.severity**? Rhapsody tried to enforce good programming practices and one of these practices is that you should always go through functions to access the data. To that end, by default, Rhapsody generates both an *accessor* (get + variable name; also known as a *getter*) and *mutator* (set + variable name; also known as a *setter*) for you. By default, *even though the visibility of the variables in Rhapsody is declared as public*, the actual variable itself is declared protected and the accessor and mutator are declared as public.

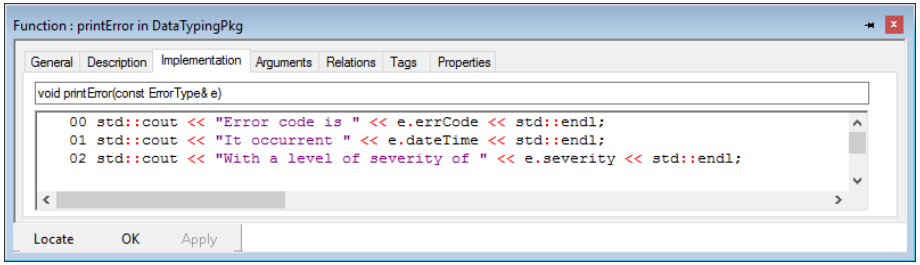
You wouldn't be the first person to be confused by this.

This behavior can be changed with properties. If you select the class, open its features dialog, go to the *Properties* Pane, select *View All*, and go to the topic *CG_CPP > Attribute > Visibility*. Here you have a drop down list. The default visibility is set to *protected*, but you can select *fromAttribute*. If you

make that change, then you can directly get and set attributes without using the accessor and mutator operations. You can even set this at the project level if you want that behavior for all classes and blocks.



If you make this change for the **ErrorType** class, then you could implement the **printError** differently.



12.1.1 Special Case: #define

It is a good programming practice to give important numbers explicit and meaningful names. Obscure numbers that just show up, unexplained, in specifications and code are often called “magic numbers”. It is not a complement. It means that there is no support to help others (or even yourself) to figure out why the number is there.

As a very simple example, consider converting foot-pounds of force to horsepower. You could write something like

```
qxt = tqp * 5252;
```

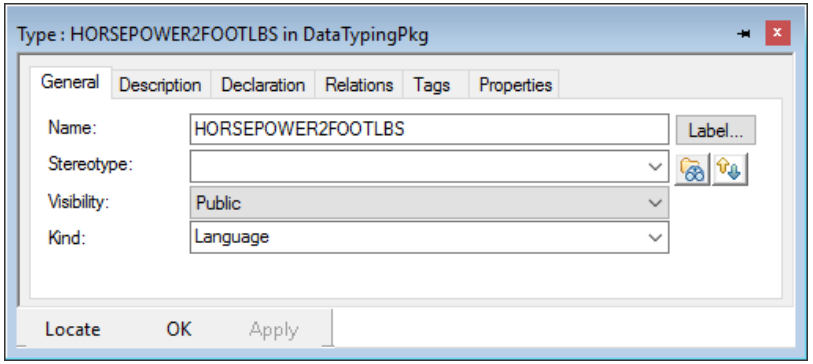
Or you could write

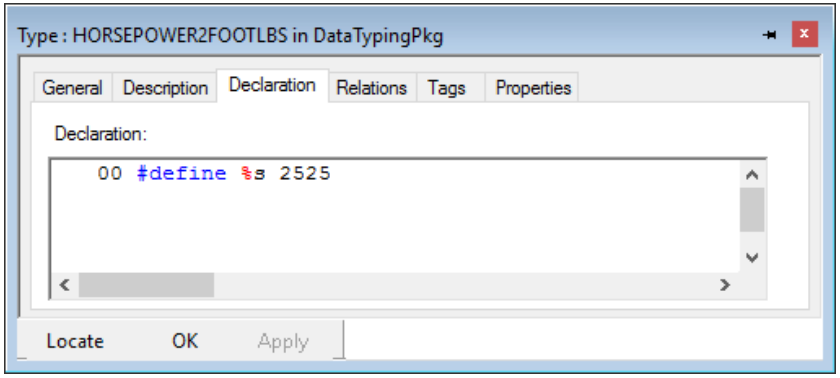
```
#define HORSEPOWER2FOOTLBS 5252
force_ftlb = force_hp *  HORSEPOWER2FOOTLBS;
```

The first line defines a constant. It is nothing more than a textual name given to a value and can be used anywhere that the value it represents can be used. It makes eases understanding of the code you write. It’s important to understand that **HORSEPOWER2FOOTLBS** is not a variable with the value of 5252. It is just another name for that value.

The second line is just an example of using meaningful names for variables; in this case, appending an abbreviation for the units in the name itself.

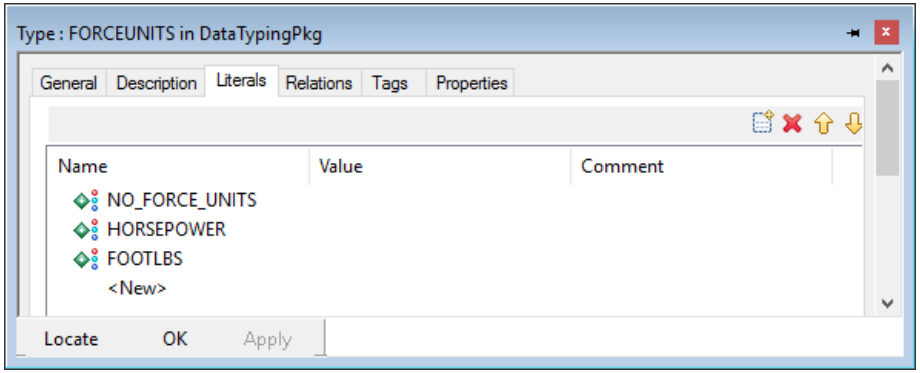
To define a named constant, add a new type, give it the kind of *Language*, and define it using *%s* to reference the value:



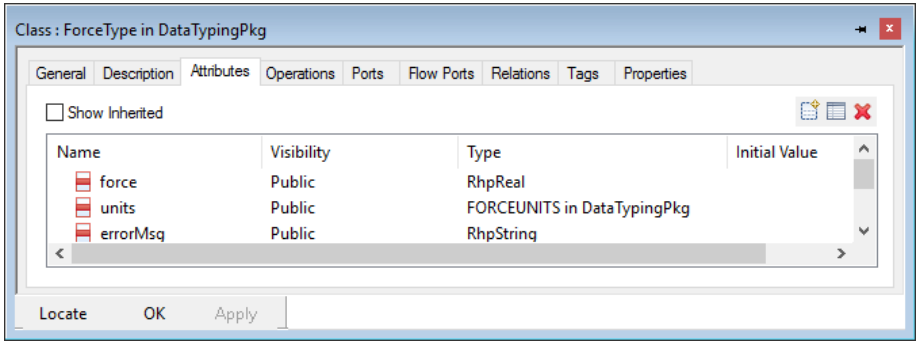


InOut (Input and Output) Parameters
InOut parameters allow you to both pass in a value and receive an updated value back. These are implemented as non-const references.

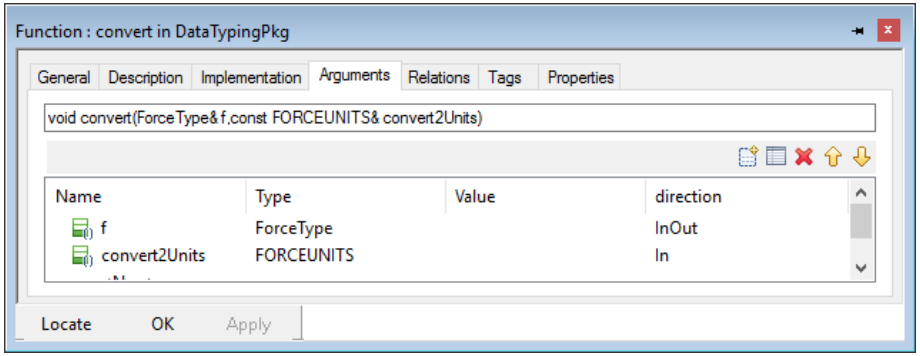
An example of this, we're going to create a **convert** operation that can convert force units between horsepower and foot-pounds. First, let's define an enumeration **FORCEUNITS** for the units. Add a new *Type* (or *DataType* if you're using SysML) of kind *Enumeration*, and define the literals:



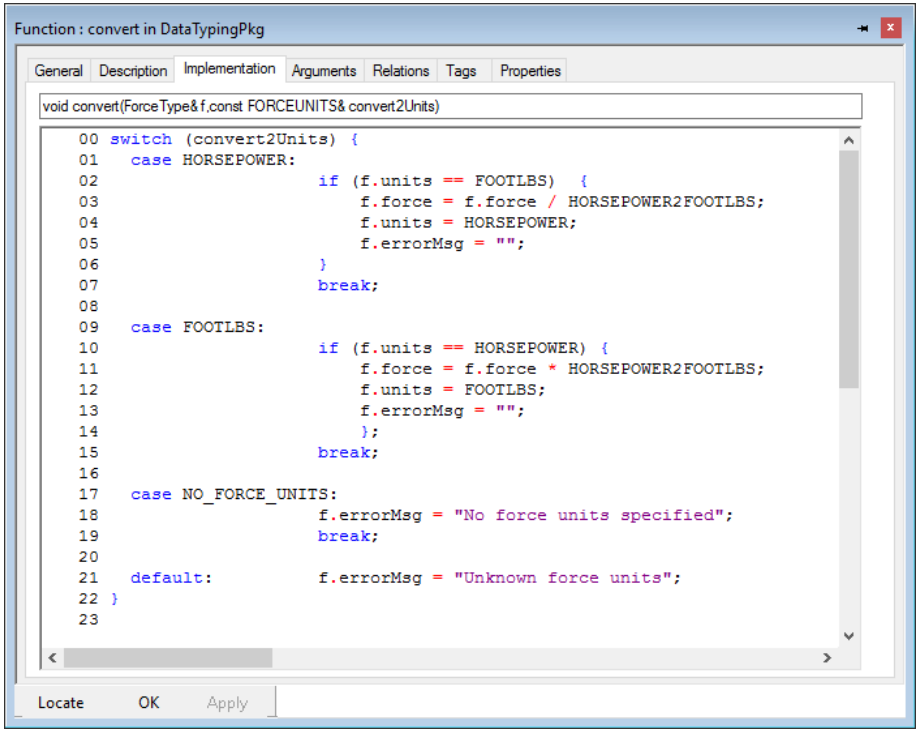
Next, let's define the type we'll be passing around, **ForceType**:



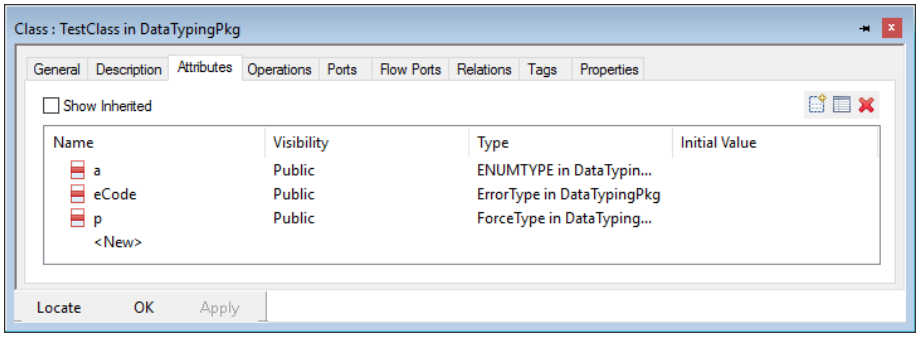
Now define the **convert** function to take two parameters; an *InOut* parameter passing in the original value and returning the converted value and an *in* parameter of the type to which to convert the incoming value.



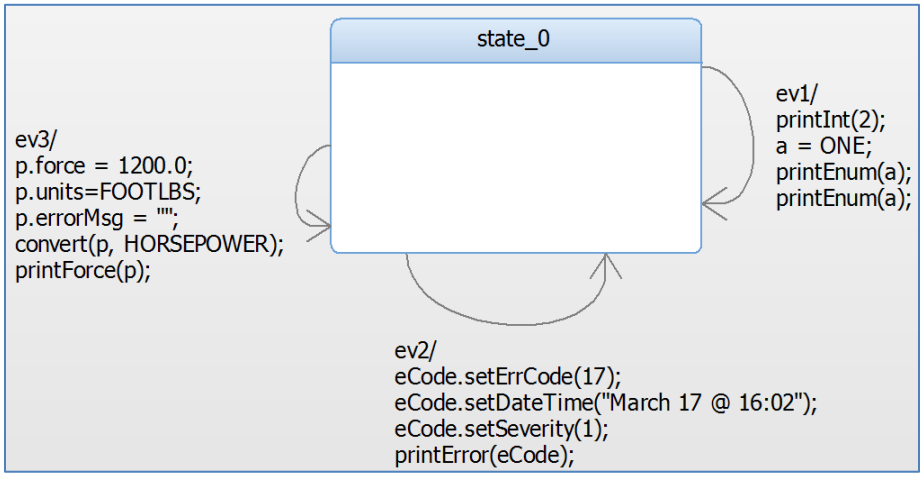
Here's the implementation of the function:



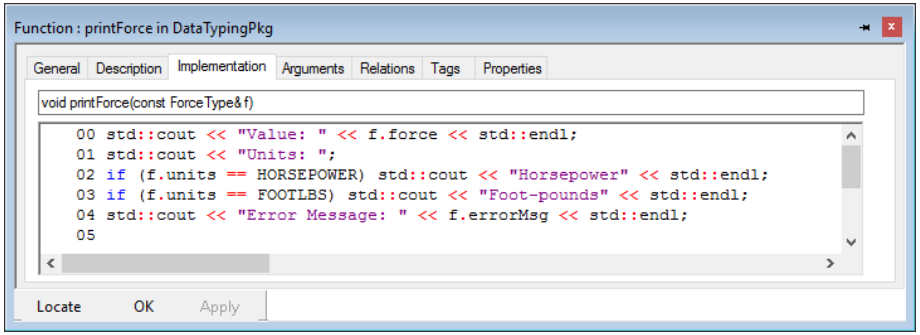
Here is a test class with attributes and a state machine to demonstrate the output of this function.



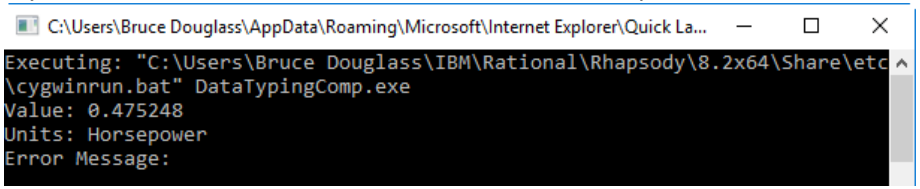
This is the state machine to support the execution of these functions:



Where **printForce** function is defined:



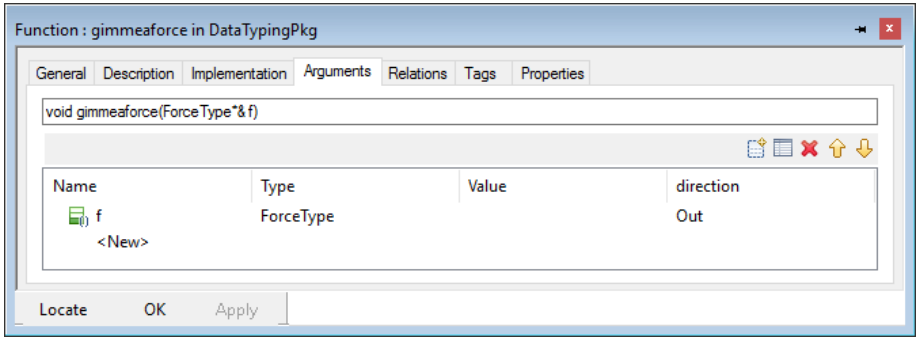
If you run it and insert the **ev3** event, this has the output:



Out (Output only) Parameters

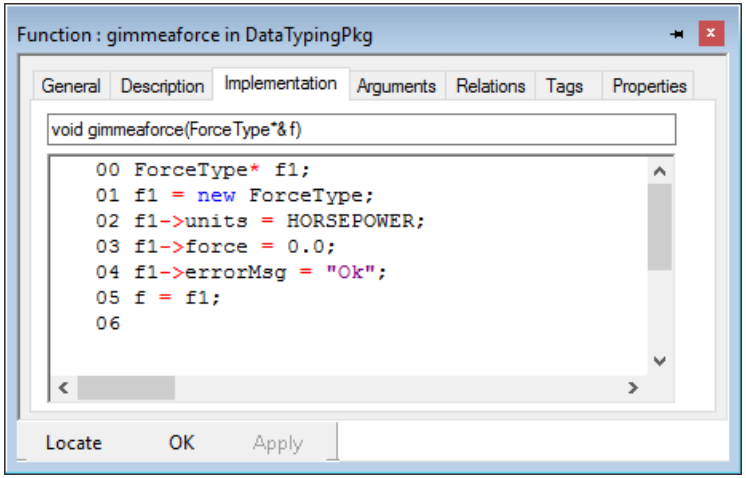
Finally, *out* parameters don't provide an input value but they do provide an output. The implementation provides a reference pointer.

Let's demonstrate that by defining a new function that returns a **ForceType**, as defined above:

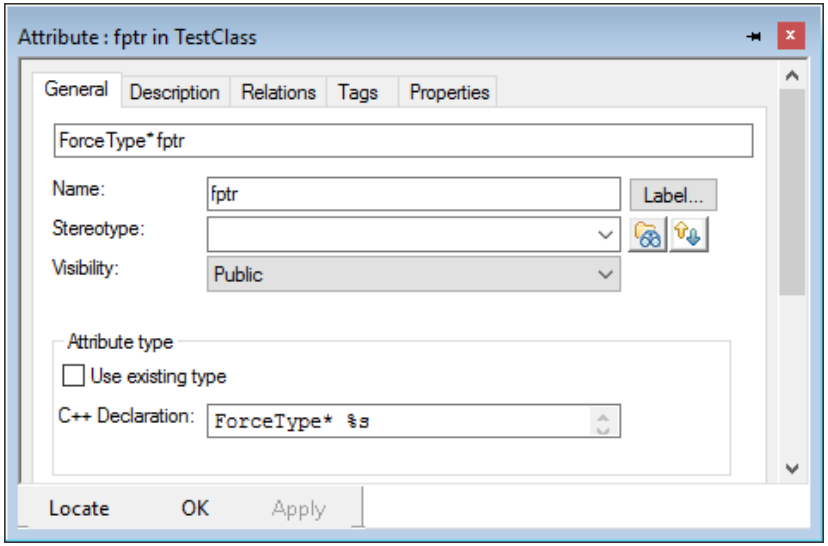


Note that the *out* parameter **f**, is defined as a pointer to a reference. This is a little bit more complex.

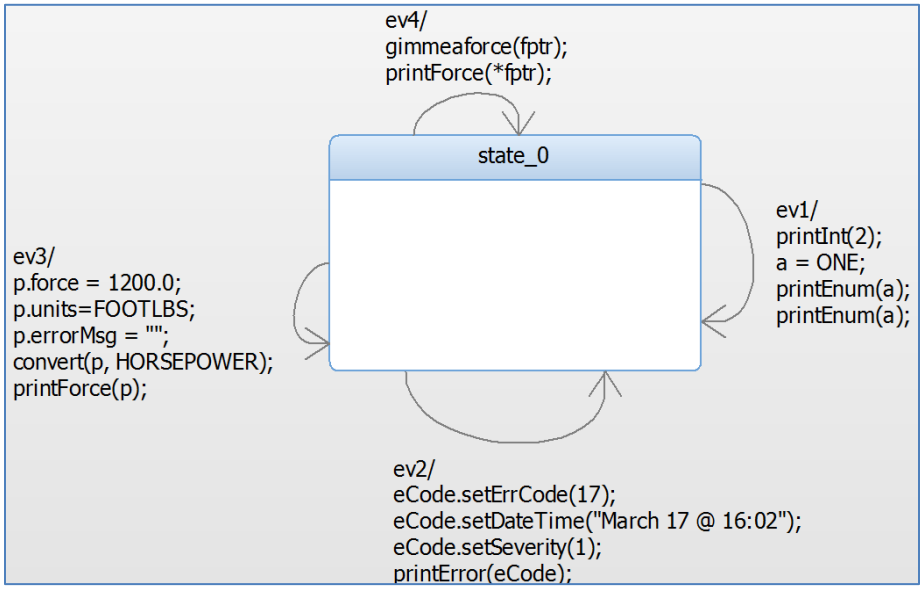
Here's the implementation. It uses a local variable, **f1**, which is a pointer to a **ForceType**. Creates a new one of them and then assigns the out parameter **f** to the value of the pointer (so now **f** points to the newly created value).



To use it, we must pass a pointer in the parameter list. So we'll add **fptr** (a pointer to a **ForceType**) to **TestClass**:



and we'll update the **TestEventClass1's** state machine to use it (see ev4).



So the function **gimmeaforce** is provided a pointer argument which is updated with a value. You can see it is dereferenced and passed in the call to **printForce**. (The expression ***fptr** returns the thing to which **fptr** points).

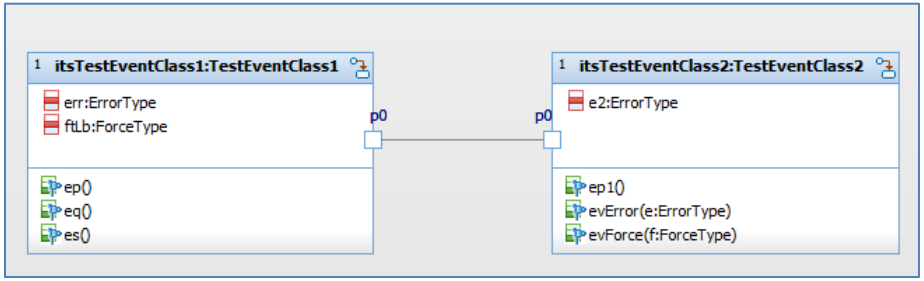
12.2 Passing Arguments in Event Receptions

So far, all the examples we've given were functions. It is similar for event receptions. Once difference is that only input parameters are supported for event receptions. If you want to be tricky and allow the state machine to return values, you can send pointer types and have the target state machine modify the values through dereferencing the pointers. More commonly, separate events are used to send return values[21] when necessary.

Another difference is that while as many parameters can be passed as desired, Rhapsody wraps them up into a *struct* called **params**, which contains pointers to each event argument. This is only visible on the receiving side of the event exchange. The values held in the **params** structure are only valid through the completion of the state machine step in which they are defined.

On the sender side, it is common to use the *Send Action* to send the action to the target object[22]. This can either use the association role name (if associations are used) or the port name (if ports are used). To demonstrate this, I've constructed a simple model.

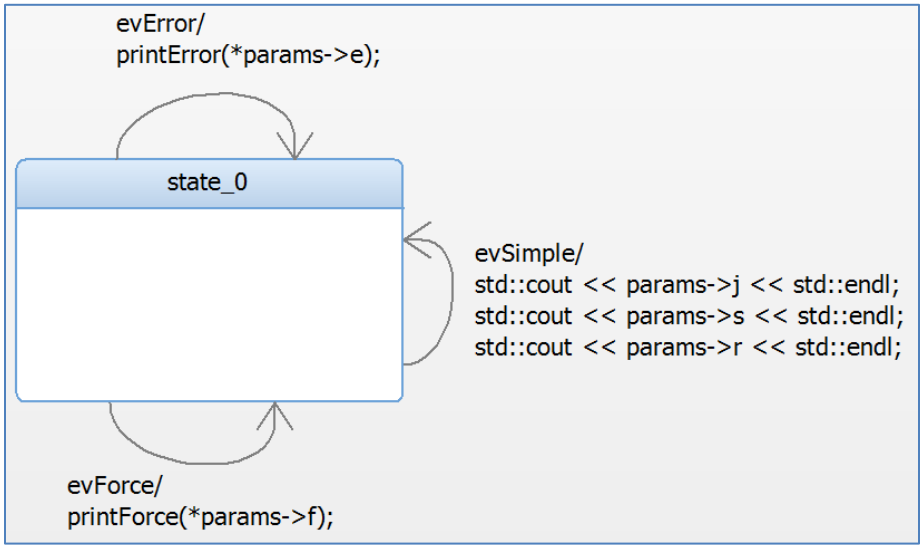
The following diagram shows the two objects connected using ports:



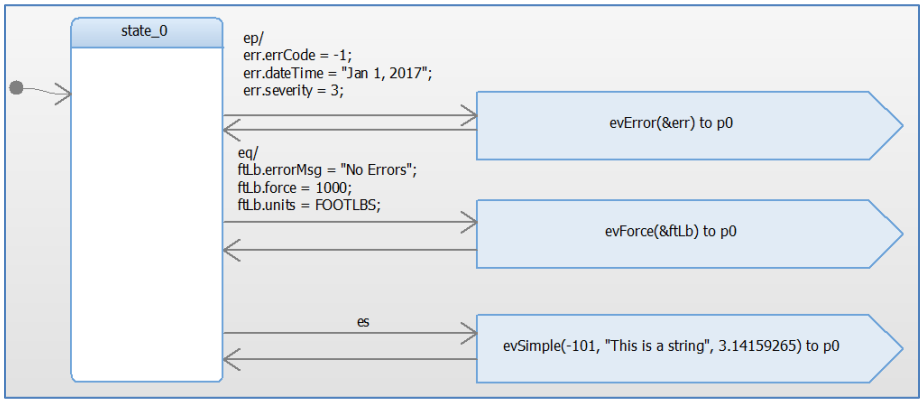
The instance of **TestEventClass2** is the event receiver in this case. Here is its state machine:

[21] Although triggered operations, a synchronous kind of event receptor, can return a value.

[22] Although the **GEN** macro is a commonly used alternative.



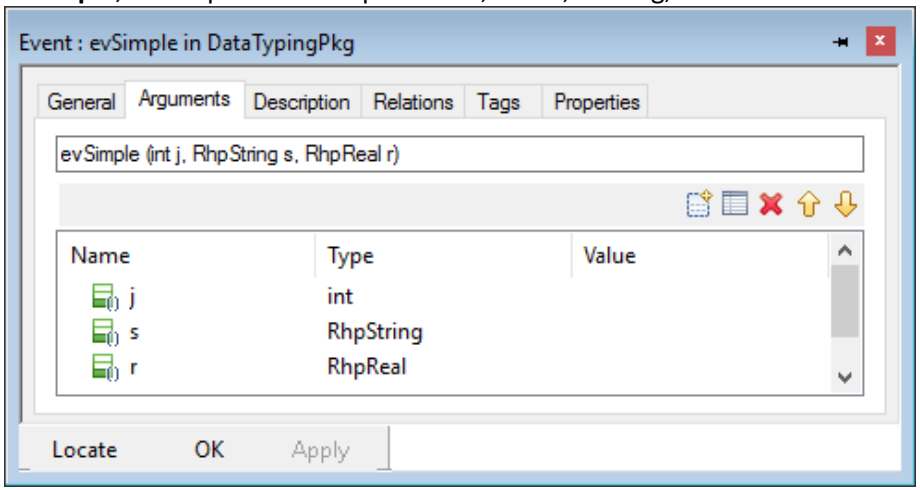
Here is the sender (**TestEventClass1**) state machine:



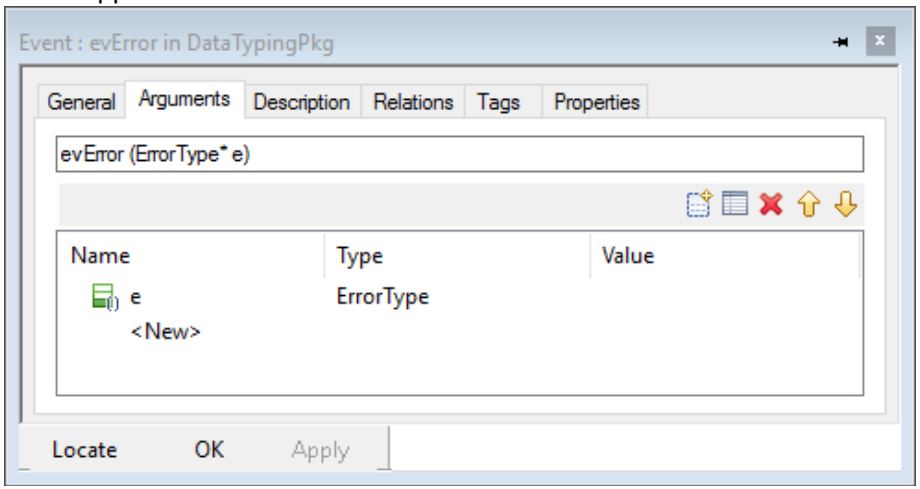
The events **ep**, **eq**, and **es** are there so that you can send them to the **TestClass1** instance to have it send a corresponding event to **TestClass2** instance.

The events of interest here are:

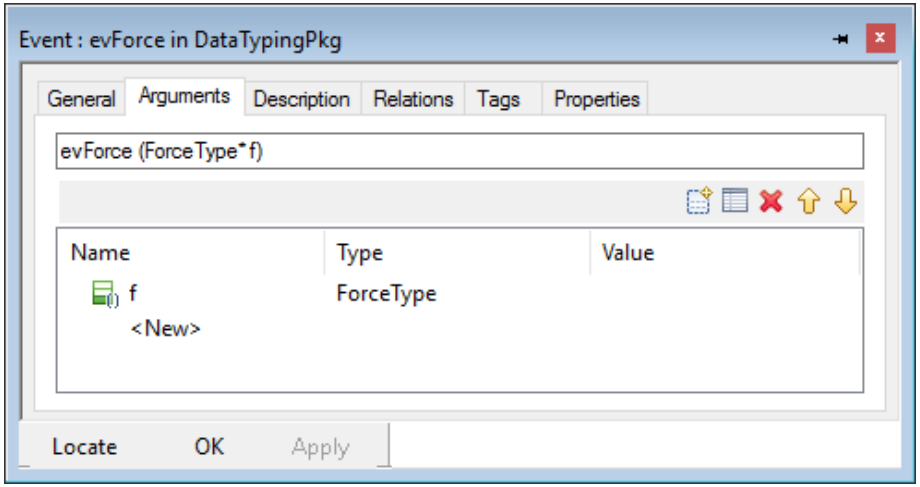
evSimple, which passes 3 simple values, an int, a string, and a real:



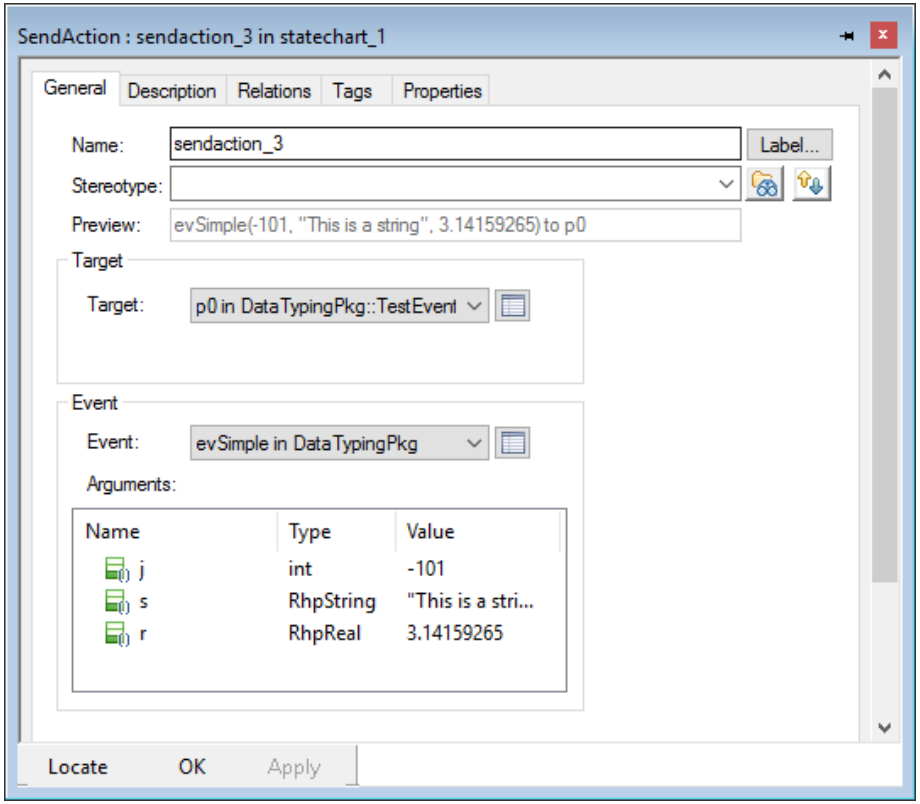
evError which passes an argument of the **ErrorType** class we defined earlier in this appendix:



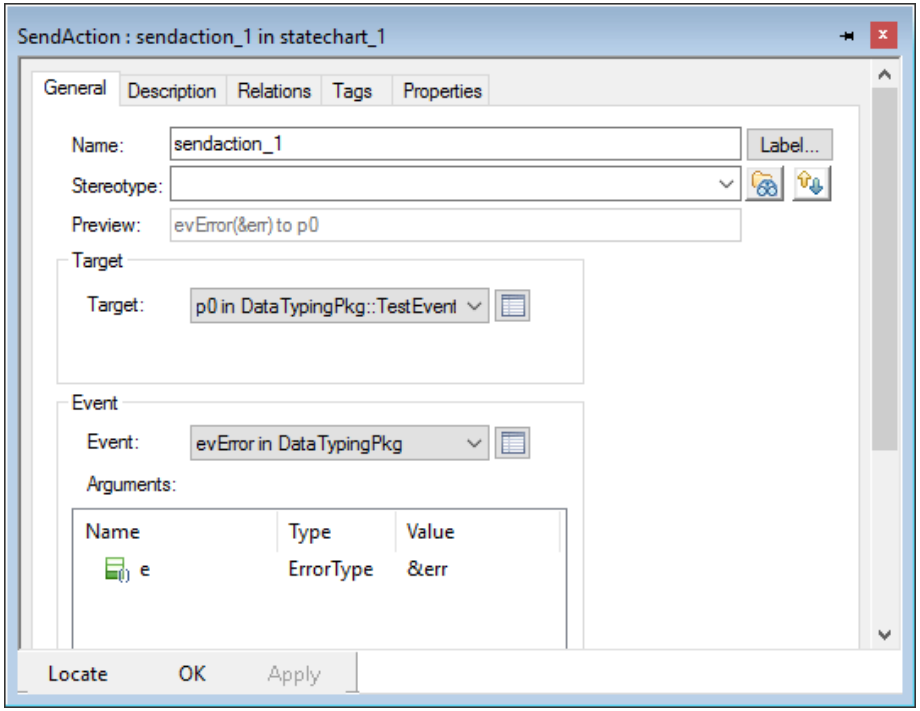
and **evForce**, which passes an argument of type **ForceType**, also defined earlier:



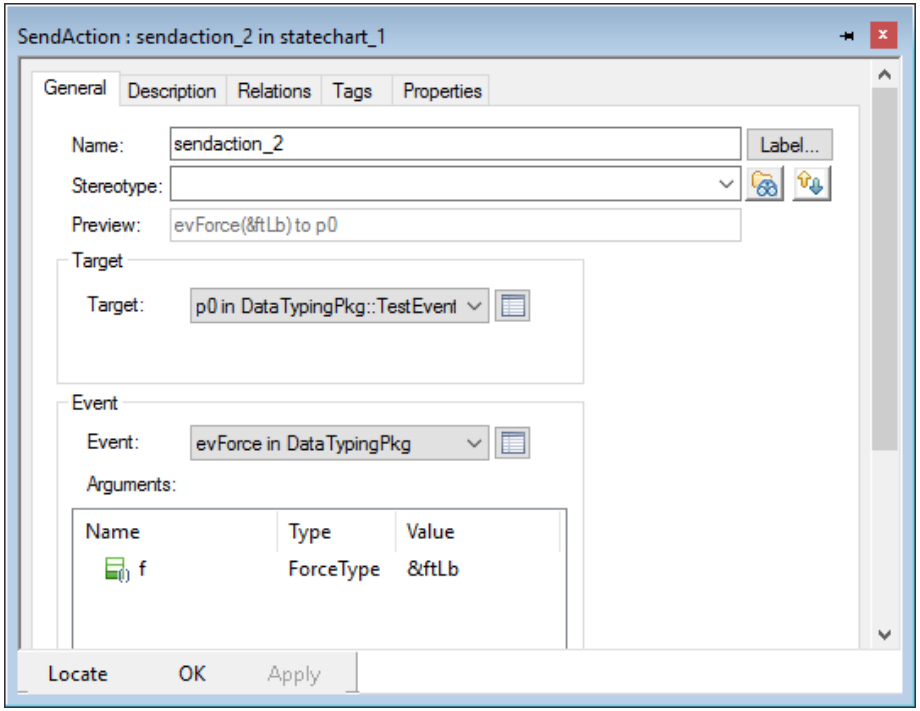
On the sender side, when we send the event, we must specify where it goes (in this case, port **p0**), and the values. For **evSimple**, the send action looks like this:



For the **evError** event, we define an attribute **err** in **TestEventClass1** of type **ErrorType** and we'll pass this. Note the use of the **&** operator to pass the address of the attribute.



Similarly for **evForce**, we define an attribute **ftLb** of type **ForceType** and pass this value.

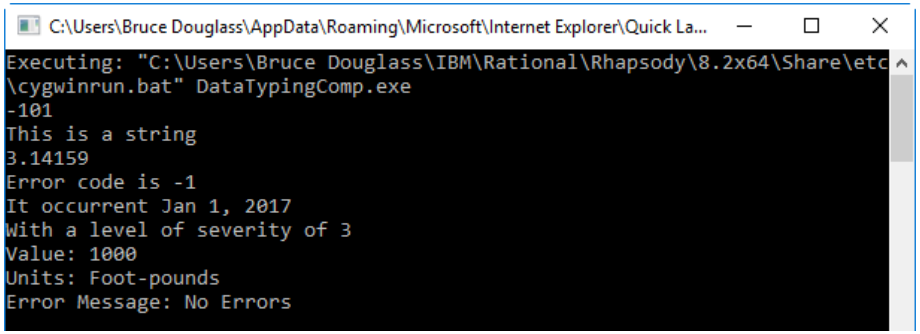


In the **TestEventClass1** state machine, we assign values to the fields for the structured types before sending the events. In the **TestEventClass2**, we use the previously defined **printError** and **printForce** functions to print the received values. Note that both these functions expect a reference to the structure type and what we have is a pointer, so we must dereference the pointer to pass it; for example, to send the parameter **e** of the **evError** event to the **printError** function we use the syntax

```
printError(*params->e);
```

For the simple parameter (see **evSimple** event), we can just deference the params structure to access the values.

The output from this model, if we invoke **evSimple**, **evError** and **evForce** looks like this:



12.3 Summary

That's pretty much it. Functions, including operations of classes and blocks, can have input, output and input/output arguments. Simple types are passed by copy but complex arguments, including enumerations, are passed by reference. Event receptions have only input arguments. Behind the scenes, Rhapsody constructs a **params** *struct* to hold pointers to the pass values. The pointers in the **params** *struct* must then be dereferenced to access the passed values.

13 Tables

This section contains a few of the larger tables from the model.

13.1 Derived Requirements Table

Requirement Name	Specification	Derived From
ACES_SS_requirement_32	Any subsystem running software shall - both at start up and upon command - run an integrity check of the installed software object code verified by a method at least as robust as 32-bit CRC check	StartUpReq_4
ACES_SS_requirement_33	Any subsystem running software that contains configuration data shall - both at start up and upon command - run an integrity check of the installed configuration verified by a method at least as robust as 32-bit CRC check as well as reasonable range checks.	StartUpReq_4
ACES_SS_requirement_34	All subsystems other than the ACES_Management subsystem shall report error status and BIT results upon query or upon completion of tests.	StartUpReq_4
ACSCUNT_requirement_10	The accuracy of movement of the control surface shall be +/- 0.5 degrees angle of +/- 0.5 cm distance.	FuncReq_36
ACSCUNT_requirement_11	Each control surface shall measure achieved control position with an accuracy of +/- 0.05 degrees or +/- 0.05 cm	FuncReq_36
ACSCUNT_requirement_12	If achieved position of any control surface unit is out of specification or takes longer than 3.0s, the control surface unit shall inform ACES_Management of the error	FuncReq_40
ACSCUNT_requirement_13	Each control surface shall accept a command for it's position and will respond with both current commanded position and current measured position.	FuncReq_40
ACSCUNT_requirement_16	The ACES_Management subsystem shall check that each command movement takes place within 3.0seconds.	FuncReq_37
ACSCUNT_requirement_17	The ACES_Management subsystem shall check that each angular movement of less than 10 degrees is performed in less than 1.0 seconds.	FuncReq_37
ACSCUNT_requirement_18	Each control surface subsystem shall report movement completion to the ACES_Management subsystem with acquired measured position and time required for the movement.	FuncReq_37
ACSCUNT_requirement_19	The ACES_Management subsystem shall listen for life ticks from each surface control subsystem interface, expecting them to arrive at least every 0.5s.	FuncReq_39
ACSCUNT_requirement_20	If the ACES_Management subsystem does not receive a life tick within 0.5s of the initiating life tick, it shall report an error to both the Pilot Display and Attitude Management systems.	FuncReq_39
ACSCUNT_requirement_21	Each control surface input shall issue a life tick message to the ACES_Management subsystem at least every 0,5s.	FuncReq_39
ACSCUNT_requirement_24	Each control surface unit instance shall have a unique identifier which shall be used to in messages to the ACES_Management subsystem.	InterfaceReq_0
ACSCUNT_requirement_25	Each control surface unit shall have, as persistent configuration data, low and high movement limits, required measurement accuracy, and movement time limits.	FuncReq_0
ACSCUNT_requirement_26	Each surface control unit instance shall report an error to the ACES_Management subsystem if the result of a commanded movement is out of specification either in accuracy or timing.	FuncReq_36
ACSCUNT_requirement_3	All control surfaces shall accept commands from the ACES_Management subsystem to set rotational position.	FuncReq_0
ACSCUNT_requirement_7	Each control surface shall accept a command to move it to the desired position and shall begin movement based on that command within 0.1 seconds.	FuncReq_0
AM_requirement_1	The ACES_Management systen shall command each control surface position either as a response to a received command or turning built in test.	FuncReq_0
AM_requirement_27	The ACES_Management subsystem shall issue an error message to the Attitude Management system if both incoming and outgoing hydraulic pressures not are within +/1 1000 kPa of the default pressure of 35000 kPa and if this is not true.	ErrorReq_34

AM_requirement_27	The ACES_Management subsystem shall issue an error message to the Attitude Management system if both incoming and outgoing hydraulic pressures not are within +/1 1000 kPa of the default pressure of 35000 kPa and if this is not true.	ErrorReq_35
AM_requirement_28	The ACES_Management subsystem shall check hydraulic pressure at least once every 2.0 seconds.	ErrorReq_35
AM_requirement_28	The ACES_Management subsystem shall check hydraulic pressure at least once every 2.0 seconds.	ErrorReq_34
AM_requirement_29	The ACES_Management subsystem shall issue an error message to the Attitude Management subsystem.if incoming or internal power for fluctuations of more than 5% in voltage.	ErrorReq_37
AM_requirement_29	The ACES_Management subsystem shall issue an error message to the Attitude Management subsystem if incoming or internal power for fluctuations of more than 5% in voltage.	ErrorReq_36
AM_requirement_30	The ACES_Management subsystem shall issue an error to the Attitude Control System within 0.5s if it detects a sudden power loss.	ErrorReq_36
AM_requirement_30	The ACES_Management subsystem shall issue an error to the Attitude Control System within 0.5s if it detects a sudden power loss.	ErrorReq_37
AM_requirement_35	The ACES_Management subsystem shall request a built in test run by every subsystem that contains software.	StartUpReq_4
AM_requirement_4	The ACES_Management subsystem shall range check each movement command for each control surface movement to ensure that the set position is in range.	FuncReq_36
AM_requirement_6	If a movement position is out of range for the a specified control surface, the ACES_Management subsystem shall reject all positions specified within the incoming command and respond with a message indicating its rejection.	FuncReq_36
AM_requirement_9	The setting precision of the ACES_Management subsystem for control surface position shall be +/- 0.1 degrees of angle or +/1 0,1 cm distance	FuncReq_36
DerConfigReq_1	Each control surface unit shall be support configuration to set min and max positions, hydraulic and power inputs and error limits, and zero position.	ConfigReq_0
DerConfigReq_2	Each control surface unit shall provide the ability to respond to requests for current configuration settings.	ConfigReq_2

DerFunReq_1	Once a each control surface has achieved its commanded position, it shall maintain station keeping adjustments to keep it within 0.1 degrees of angle or 0.1cm of extension, as appropriate, at least 10 times per second.	FuncReq_36
DerIntReq_1	The Control Surface subsystem types shall provide an interface to set and get the commanded control surface position.	InterfaceReq_0
DerIntReq_10	Each control surface subsystem shall detect faults and report them to the ACES Management subsystem.	InterfaceReq_3
DerIntReq_15	The ACES Power system shall distribute power from the aircraft to the ACES internal subsystems.	InterfaceReq_4
DerIntReq_16	The ACES Power subsystem shall provide an interface to select input source.	InterfaceReq_5
DerIntReq_17	The ACES Power subsystem shall monitor incoming current and voltage and inform the ACES Management system if the current or voltage exceeds nominal values by more than 10% for more than 30 seconds, or by more than 30% for more than 2 seconds.	InterfaceReq_5
DerIntReq_18	The ACES Management system will monitor the power from the ACES Power subsystem and automatically switch if it receives a power fault,.	InterfaceReq_5
DerIntReq_2	The Control Surface subsystem types shall provide an interface to get the measured control surface position.	InterfaceReq_0
DerIntReq_3	The Control Surface With Trim subsystem type shall provide an interface to set and get the commanded trim tab control surface position.	InterfaceReq_0
DerIntReq_4	The Control Surface With Trim subsystem type shall provide an interface to get the measured trim tab control surface position.	InterfaceReq_0
DerIntReq_5	Retracting control surface subsystem type shall provide an interface to set and get the commanded extension of the control surface.	InterfaceReq_0
DerIntReq_6	Retracting control surface subsystem type shall provide an interface to get the control surface measured extension.	InterfaceReq_0

DerIntReq_7	Every second the ACES Management subsystem will query all the control surface measures positions and relay them to the Attitude Management System	InterfaceReq_1
DerIntReq_8	The control surfaces subsystems shall provide an interface to request their hydraulic and power status.	InterfaceReq_2
DerIntReq_9	The ACES Management system shall provide the status of power and hydraulics to the pilot display at least every second while operational.	InterfaceReq_2
DerStartUpReq_1	Each control surface unit shall support a Built In Test (BIT) that is only available while not operational, for checking movement ranges, accuracy, and timing.	StartUpReq_4
DerStartupReq_2	Each control surface unit shall support periodic BIT (PBIT) run at least every 30 seconds; this test suite shall only run tests which do not interfere with surface control operation.	StartUpReq_4
DerStartupReq_3	All BIT and PBIT results from the Control Surface subsystem shall be reported to the ACES Management System.	StartUpReq_4
DerStartupReq_4	The ACES Management system shall maintain system state	StartUpReq_4

Table 4: Derived Requirements Table (Complete)

13.2 Subsystem Requirements Allocation Table

Package	Subsystem	Requirement
ACESDecompositionPkg		
ACES_Control_SurfacePkg		
ACES_Control_SurfacePkg	ACES_Control_Surface	ACES_SS_requirement_32
ACES_Control_SurfacePkg	ACES_Control_Surface	ACES_SS_requirement_33
ACES_Control_SurfacePkg	ACES_Control_Surface	ACES_SS_requirement_34
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_10
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_11
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_12
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_13
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_18
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_19
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_21
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_24
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_25
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_26
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_3
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_7
ACES_Control_SurfacePkg	ACES_Control_Surface	ConfigReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	ConfigReq_3
ACES_Control_SurfacePkg	ACES_Control_Surface	DerConfigReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	DerConfigReq_2
ACES_Control_SurfacePkg	ACES_Control_Surface	DerFunReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_10
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_11
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_12
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_14
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_2
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_8
ACES_Control_SurfacePkg	ACES_Control_Surface	DerReqInt_13
ACES_Control_SurfacePkg	ACES_Control_Surface	DerStartUpReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	DerStartupReq_2
ACES_Control_SurfacePkg	ACES_Control_Surface	DerStartupReq_3
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_26
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_27
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_28
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_29
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_3
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_34
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_35
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_36
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_37

ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_25
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_27
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_28
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_29
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_30
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_36
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_37
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_40
ACES_Control_SurfacePkg	ACES_Control_Surface	OtherReq_0
ACES_Control_SurfacePkg	ACES_Control_Surface	OtherReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	SafetyReq_006
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390202
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390207
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390209
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390210
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390211
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390212
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390213
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390214
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390215
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390217
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390218
ACES_Control_SurfacePkg	ACES_Control_Surface	ACES_SS_requirement_32
ACES_Control_SurfacePkg	ACES_Control_Surface	ACES_SS_requirement_33
ACES_Control_SurfacePkg	ACES_Control_Surface	ACES_SS_requirement_34
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_10
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_11
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_12
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_13
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_18
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_19
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_21
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_24
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_25
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_26
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_3
ACES_Control_SurfacePkg	ACES_Control_Surface	ACSCUNT_requirement_7
ACES_Control_SurfacePkg	ACES_Control_Surface	ConfigReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	ConfigReq_3
ACES_Control_SurfacePkg	ACES_Control_Surface	DerConfigReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	DerConfigReq_2
ACES_Control_SurfacePkg	ACES_Control_Surface	DerFunReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_10
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_11
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_12
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_14
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_2
ACES_Control_SurfacePkg	ACES_Control_Surface	DerIntReq_8
ACES_Control_SurfacePkg	ACES_Control_Surface	DerReqInt_13
ACES_Control_SurfacePkg	ACES_Control_Surface	DerStartUpReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	DerStartupReq_2
ACES_Control_SurfacePkg	ACES_Control_Surface	DerStartupReq_3
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_26
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_27
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_28
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_29
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_3
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_34
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_35
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_36
ACES_Control_SurfacePkg	ACES_Control_Surface	ErrorReq_37
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_25
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_27

ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_28
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_29
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_30
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_36
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_37
ACES_Control_SurfacePkg	ACES_Control_Surface	FuncReq_40
ACES_Control_SurfacePkg	ACES_Control_Surface	OtherReq_0
ACES_Control_SurfacePkg	ACES_Control_Surface	OtherReq_1
ACES_Control_SurfacePkg	ACES_Control_Surface	SafetyReq_006
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390202
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390207
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390209
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390210
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390211
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390212
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390213
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390214
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390215
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390217
ACES_Control_SurfacePkg	ACES_Control_Surface	Safety_Req_390218
ACES_Control_Surface_RetractingPkg		
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACES_SS_requirement_32
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACES_SS_requirement_33
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACES_SS_requirement_34
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_10
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_11
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_12
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_13
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_18
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_19
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_21
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_24
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_25
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_26
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	AM_requirement_35
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ConfigReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ConfigReq_3
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerConfigReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerConfigReq_2
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerFunReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_10
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_11
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_12
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_14
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_2
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_8
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerReqInt_13
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerStartUpReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerStartupReq_2
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerStartupReq_3
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_3
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_30
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_31
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_32
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_33
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_34
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_35
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_36
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_37
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	FuncReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	FuncReq_36
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	FuncReq_37
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	FuncReq_40

ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	OtherReq_0
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	OtherReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	SafetyReq_006
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390202
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390203
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390204
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390207
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390209
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390210
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390211
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390212
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390213
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390214
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390215
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390217
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390218
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACES_SS_requirement_32
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACES_SS_requirement_33
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACES_SS_requirement_34
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_10
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_11
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_12
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_13
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_18
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_19
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_21
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_24
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_25
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ACSCUNT_requirement_26
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	AM_requirement_35
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ConfigReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ConfigReq_3
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerConfigReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerConfigReq_2
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerFunReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_10
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_11
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_12
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_14
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_2
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerIntReq_8
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerReqInt_13
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerStartUpReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerStartupReq_2
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	DerStartupReq_3
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_3
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_30
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_31
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_32
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_33
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_34
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_35
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_36
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	ErrorReq_37
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	FuncReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	FuncReq_36
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	FuncReq_37
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	FuncReq_40
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	OtherReq_0
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	OtherReq_1
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	SafetyReq_006
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390202
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390203

ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390204
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390207
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390209
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390210
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390211
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390212
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390213
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390214
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390215
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390217
ACES_Control_Surface_RetractingPkg	ACES_Control_Surface_Retracting	Safety_Req_390218
ACES_Control_Surface_With_TrimPkg		
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACES_SS_requirement_32
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACES_SS_requirement_33
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACES_SS_requirement_34
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_10
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_11
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_12
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_13
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_18
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_19
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_21
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_24
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_25
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_26
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_7
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	AM_requirement_35
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ConfigReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ConfigReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerConfigReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerConfigReq_2
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerFunReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_10
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_11
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_12
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_14
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_2
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_4
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_8
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerReqInt_13
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerStartUpReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerStartupReq_2
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerStartupReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_10
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_11
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_12
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_13
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_14
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_15
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_16
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_17
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_18
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_19
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_20
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_21
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_22
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_23
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_24
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_25
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_34

ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_35
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_36
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_37
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_4
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_5
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_6
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_7
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_8
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_9
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_10
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_11
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_12
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_13
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_15
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_16
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_17
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_18
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_19
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_2
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_20
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_21
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_22
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_23
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_24
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_25
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_26
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_35
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_36
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_37
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_4
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_40
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_5
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_6
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_7
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_8
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_9
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	OtherReq_0
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	OtherReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	SafetyReq_006
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390202
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390203
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390204
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390207
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390209
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390210
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390211
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390212
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390213
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390214
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390215
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390217
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390218
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACES_SS_requirement_32
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACES_SS_requirement_33
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACES_SS_requirement_34
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_10
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_11
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_12
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_13
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_18
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_19
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_21
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_24

ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_25
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_26
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ACSCUNT_requirement_7
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	AM_requirement_35
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ConfigReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ConfigReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerConfigReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerConfigReq_2
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerFunReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_10
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_11
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_12
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_14
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_2
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_4
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerIntReq_8
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerReqInt_13
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerStartUpReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerStartupReq_2
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	DerStartupReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_10
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_11
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_12
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_13
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_14
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_15
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_16
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_17
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_18
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_19
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_20
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_21
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_22
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_23
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_24
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_25
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_34
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_35
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_36
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_37
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_4
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_5
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_6
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_7
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_8
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	ErrorReq_9
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_10
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_11
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_12
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_13
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_15
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_16
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_17
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_18
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_19
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_2
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_20
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_21
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_22
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_23

ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_24
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_25
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_26
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_3
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_35
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_36
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_37
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_4
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_40
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_5
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_6
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_7
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_8
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	FuncReq_9
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	OtherReq_0
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	OtherReq_1
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	SafetyReq_006
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390202
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390203
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390204
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390207
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390209
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390210
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390211
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390212
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390213
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390214
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390215
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390217
ACES_Control_Surface_With_TrimPkg	ACES_Control_Surface_With_Trim	Safety_Req_390218
ACES_HydraulicsPkg		
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_001
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_002
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_004
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_005
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_390197
ACES_HydraulicsPkg	ACES_Hydraulics	Safety_Req_390198
ACES_HydraulicsPkg	ACES_Hydraulics	Safety_Req_390200
ACES_HydraulicsPkg	ACES_Hydraulics	Safety_Req_390201
ACES_HydraulicsPkg	ACES_Hydraulics	Safety_Req_390209
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_001
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_002
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_004
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_005
ACES_HydraulicsPkg	ACES_Hydraulics	SafetyReq_390197
ACES_HydraulicsPkg	ACES_Hydraulics	Safety_Req_390198
ACES_HydraulicsPkg	ACES_Hydraulics	Safety_Req_390200
ACES_HydraulicsPkg	ACES_Hydraulics	Safety_Req_390201
ACES_HydraulicsPkg	ACES_Hydraulics	Safety_Req_390209
ACES_ManagementPkg		
ACES_ManagementPkg	ACES_Management	ACES_SS_requirement_32
ACES_ManagementPkg	ACES_Management	ACES_SS_requirement_33
ACES_ManagementPkg	ACES_Management	ACES_SS_requirement_34
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_12
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_13
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_16
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_17
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_18
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_19
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_20
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_21
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_24
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_26
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_3

ACES_ManagementPkg	ACES_Management	AM_requirement_1
ACES_ManagementPkg	ACES_Management	AM_requirement_27
ACES_ManagementPkg	ACES_Management	AM_requirement_28
ACES_ManagementPkg	ACES_Management	AM_requirement_29
ACES_ManagementPkg	ACES_Management	AM_requirement_30
ACES_ManagementPkg	ACES_Management	AM_requirement_35
ACES_ManagementPkg	ACES_Management	AM_requirement_4
ACES_ManagementPkg	ACES_Management	AM_requirement_6
ACES_ManagementPkg	ACES_Management	AM_requirement_9
ACES_ManagementPkg	ACES_Management	ConfigReq_1
ACES_ManagementPkg	ACES_Management	ConfigReq_3
ACES_ManagementPkg	ACES_Management	DerIntReq_11
ACES_ManagementPkg	ACES_Management	DerIntReq_12
ACES_ManagementPkg	ACES_Management	DerIntReq_14
ACES_ManagementPkg	ACES_Management	DerIntReq_16
ACES_ManagementPkg	ACES_Management	DerIntReq_17
ACES_ManagementPkg	ACES_Management	DerIntReq_18
ACES_ManagementPkg	ACES_Management	DerIntReq_7
ACES_ManagementPkg	ACES_Management	DerIntReq_9
ACES_ManagementPkg	ACES_Management	DerReqInt_13
ACES_ManagementPkg	ACES_Management	DerStartupReq_4
ACES_ManagementPkg	ACES_Management	ErrorReq_0
ACES_ManagementPkg	ACES_Management	ErrorReq_1
ACES_ManagementPkg	ACES_Management	ErrorReq_10
ACES_ManagementPkg	ACES_Management	ErrorReq_11
ACES_ManagementPkg	ACES_Management	ErrorReq_12
ACES_ManagementPkg	ACES_Management	ErrorReq_13
ACES_ManagementPkg	ACES_Management	ErrorReq_14
ACES_ManagementPkg	ACES_Management	ErrorReq_15
ACES_ManagementPkg	ACES_Management	ErrorReq_16
ACES_ManagementPkg	ACES_Management	ErrorReq_17
ACES_ManagementPkg	ACES_Management	ErrorReq_18
ACES_ManagementPkg	ACES_Management	ErrorReq_19
ACES_ManagementPkg	ACES_Management	ErrorReq_2
ACES_ManagementPkg	ACES_Management	ErrorReq_20
ACES_ManagementPkg	ACES_Management	ErrorReq_21
ACES_ManagementPkg	ACES_Management	ErrorReq_22
ACES_ManagementPkg	ACES_Management	ErrorReq_23
ACES_ManagementPkg	ACES_Management	ErrorReq_24
ACES_ManagementPkg	ACES_Management	ErrorReq_25
ACES_ManagementPkg	ACES_Management	ErrorReq_26
ACES_ManagementPkg	ACES_Management	ErrorReq_27
ACES_ManagementPkg	ACES_Management	ErrorReq_28
ACES_ManagementPkg	ACES_Management	ErrorReq_29
ACES_ManagementPkg	ACES_Management	ErrorReq_3
ACES_ManagementPkg	ACES_Management	ErrorReq_30
ACES_ManagementPkg	ACES_Management	ErrorReq_31
ACES_ManagementPkg	ACES_Management	ErrorReq_32
ACES_ManagementPkg	ACES_Management	ErrorReq_33
ACES_ManagementPkg	ACES_Management	ErrorReq_34
ACES_ManagementPkg	ACES_Management	ErrorReq_35
ACES_ManagementPkg	ACES_Management	ErrorReq_36
ACES_ManagementPkg	ACES_Management	ErrorReq_37
ACES_ManagementPkg	ACES_Management	ErrorReq_4
ACES_ManagementPkg	ACES_Management	ErrorReq_5
ACES_ManagementPkg	ACES_Management	ErrorReq_6
ACES_ManagementPkg	ACES_Management	ErrorReq_7
ACES_ManagementPkg	ACES_Management	ErrorReq_8
ACES_ManagementPkg	ACES_Management	ErrorReq_9
ACES_ManagementPkg	ACES_Management	FuncReq_0
ACES_ManagementPkg	ACES_Management	FuncReq_1
ACES_ManagementPkg	ACES_Management	FuncReq_10
ACES_ManagementPkg	ACES_Management	FuncReq_11
ACES_ManagementPkg	ACES_Management	FuncReq_12

ACES_ManagementPkg	ACES_Management	FuncReq_13
ACES_ManagementPkg	ACES_Management	FuncReq_15
ACES_ManagementPkg	ACES_Management	FuncReq_16
ACES_ManagementPkg	ACES_Management	FuncReq_17
ACES_ManagementPkg	ACES_Management	FuncReq_18
ACES_ManagementPkg	ACES_Management	FuncReq_19
ACES_ManagementPkg	ACES_Management	FuncReq_2
ACES_ManagementPkg	ACES_Management	FuncReq_20
ACES_ManagementPkg	ACES_Management	FuncReq_21
ACES_ManagementPkg	ACES_Management	FuncReq_22
ACES_ManagementPkg	ACES_Management	FuncReq_23
ACES_ManagementPkg	ACES_Management	FuncReq_24
ACES_ManagementPkg	ACES_Management	FuncReq_25
ACES_ManagementPkg	ACES_Management	FuncReq_26
ACES_ManagementPkg	ACES_Management	FuncReq_27
ACES_ManagementPkg	ACES_Management	FuncReq_28
ACES_ManagementPkg	ACES_Management	FuncReq_29
ACES_ManagementPkg	ACES_Management	FuncReq_3
ACES_ManagementPkg	ACES_Management	FuncReq_30
ACES_ManagementPkg	ACES_Management	FuncReq_31
ACES_ManagementPkg	ACES_Management	FuncReq_32
ACES_ManagementPkg	ACES_Management	FuncReq_33
ACES_ManagementPkg	ACES_Management	FuncReq_34
ACES_ManagementPkg	ACES_Management	FuncReq_35
ACES_ManagementPkg	ACES_Management	FuncReq_36
ACES_ManagementPkg	ACES_Management	FuncReq_37
ACES_ManagementPkg	ACES_Management	FuncReq_38
ACES_ManagementPkg	ACES_Management	FuncReq_39
ACES_ManagementPkg	ACES_Management	FuncReq_4
ACES_ManagementPkg	ACES_Management	FuncReq_40
ACES_ManagementPkg	ACES_Management	FuncReq_5
ACES_ManagementPkg	ACES_Management	FuncReq_6
ACES_ManagementPkg	ACES_Management	FuncReq_7
ACES_ManagementPkg	ACES_Management	FuncReq_8
ACES_ManagementPkg	ACES_Management	FuncReq_9
ACES_ManagementPkg	ACES_Management	InterfaceReq_0
ACES_ManagementPkg	ACES_Management	InterfaceReq_1
ACES_ManagementPkg	ACES_Management	InterfaceReq_10
ACES_ManagementPkg	ACES_Management	InterfaceReq_11
ACES_ManagementPkg	ACES_Management	InterfaceReq_12
ACES_ManagementPkg	ACES_Management	InterfaceReq_13
ACES_ManagementPkg	ACES_Management	InterfaceReq_14
ACES_ManagementPkg	ACES_Management	InterfaceReq_15
ACES_ManagementPkg	ACES_Management	InterfaceReq_16
ACES_ManagementPkg	ACES_Management	InterfaceReq_17
ACES_ManagementPkg	ACES_Management	InterfaceReq_18
ACES_ManagementPkg	ACES_Management	InterfaceReq_19
ACES_ManagementPkg	ACES_Management	InterfaceReq_2
ACES_ManagementPkg	ACES_Management	InterfaceReq_20
ACES_ManagementPkg	ACES_Management	InterfaceReq_21
ACES_ManagementPkg	ACES_Management	InterfaceReq_22
ACES_ManagementPkg	ACES_Management	InterfaceReq_23
ACES_ManagementPkg	ACES_Management	InterfaceReq_24
ACES_ManagementPkg	ACES_Management	InterfaceReq_25
ACES_ManagementPkg	ACES_Management	InterfaceReq_26
ACES_ManagementPkg	ACES_Management	InterfaceReq_27
ACES_ManagementPkg	ACES_Management	InterfaceReq_28
ACES_ManagementPkg	ACES_Management	InterfaceReq_29
ACES_ManagementPkg	ACES_Management	InterfaceReq_3
ACES_ManagementPkg	ACES_Management	InterfaceReq_4
ACES_ManagementPkg	ACES_Management	InterfaceReq_5
ACES_ManagementPkg	ACES_Management	InterfaceReq_6
ACES_ManagementPkg	ACES_Management	InterfaceReq_7
ACES_ManagementPkg	ACES_Management	InterfaceReq_8

ACES_ManagementPkg	ACES_Management	InterfaceReq_9
ACES_ManagementPkg	ACES_Management	OtherReq_0
ACES_ManagementPkg	ACES_Management	OtherReq_1
ACES_ManagementPkg	ACES_Management	SafetyReq_001
ACES_ManagementPkg	ACES_Management	SafetyReq_002
ACES_ManagementPkg	ACES_Management	SafetyReq_003
ACES_ManagementPkg	ACES_Management	SafetyReq_004
ACES_ManagementPkg	ACES_Management	SafetyReq_005
ACES_ManagementPkg	ACES_Management	SafetyReq_390197
ACES_ManagementPkg	ACES_Management	Safety_Req_390198
ACES_ManagementPkg	ACES_Management	Safety_Req_390199
ACES_ManagementPkg	ACES_Management	Safety_Req_390200
ACES_ManagementPkg	ACES_Management	Safety_Req_390201
ACES_ManagementPkg	ACES_Management	Safety_Req_390206
ACES_ManagementPkg	ACES_Management	Safety_Req_390207
ACES_ManagementPkg	ACES_Management	Safety_Req_390208
ACES_ManagementPkg	ACES_Management	Safety_Req_390209
ACES_ManagementPkg	ACES_Management	Safety_Req_390210
ACES_ManagementPkg	ACES_Management	Safety_Req_390212
ACES_ManagementPkg	ACES_Management	Safety_Req_390213
ACES_ManagementPkg	ACES_Management	Safety_Req_390214
ACES_ManagementPkg	ACES_Management	Safety_Req_390215
ACES_ManagementPkg	ACES_Management	Safety_Req_390216
ACES_ManagementPkg	ACES_Management	StartUpReq_1
ACES_ManagementPkg	ACES_Management	StartUpReq_2
ACES_ManagementPkg	ACES_Management	StartUpReq_3
ACES_ManagementPkg	ACES_Management	StartUpReq_4
ACES_ManagementPkg	ACES_Management	StartUpReq_5
ACES_ManagementPkg	ACES_Management	StartUpReq_6
ACES_ManagementPkg	ACES_Management	ACES_SS_requirement_32
ACES_ManagementPkg	ACES_Management	ACES_SS_requirement_33
ACES_ManagementPkg	ACES_Management	ACES_SS_requirement_34
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_12
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_13
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_16
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_17
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_18
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_19
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_20
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_21
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_24
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_26
ACES_ManagementPkg	ACES_Management	ACSCUNT_requirement_3
ACES_ManagementPkg	ACES_Management	AM_requirement_1
ACES_ManagementPkg	ACES_Management	AM_requirement_27
ACES_ManagementPkg	ACES_Management	AM_requirement_28
ACES_ManagementPkg	ACES_Management	AM_requirement_29
ACES_ManagementPkg	ACES_Management	AM_requirement_30
ACES_ManagementPkg	ACES_Management	AM_requirement_35
ACES_ManagementPkg	ACES_Management	AM_requirement_4
ACES_ManagementPkg	ACES_Management	AM_requirement_6
ACES_ManagementPkg	ACES_Management	AM_requirement_9
ACES_ManagementPkg	ACES_Management	ConfigReq_1
ACES_ManagementPkg	ACES_Management	ConfigReq_3
ACES_ManagementPkg	ACES_Management	DerIntReq_11
ACES_ManagementPkg	ACES_Management	DerIntReq_12
ACES_ManagementPkg	ACES_Management	DerIntReq_14
ACES_ManagementPkg	ACES_Management	DerIntReq_16
ACES_ManagementPkg	ACES_Management	DerIntReq_17
ACES_ManagementPkg	ACES_Management	DerIntReq_18
ACES_ManagementPkg	ACES_Management	DerIntReq_7
ACES_ManagementPkg	ACES_Management	DerIntReq_9
ACES_ManagementPkg	ACES_Management	DerReqInt_13
ACES_ManagementPkg	ACES_Management	DerStartupReq_4

ACES_ManagementPkg	ACES_Management	ErrorReq_0
ACES_ManagementPkg	ACES_Management	ErrorReq_1
ACES_ManagementPkg	ACES_Management	ErrorReq_10
ACES_ManagementPkg	ACES_Management	ErrorReq_11
ACES_ManagementPkg	ACES_Management	ErrorReq_12
ACES_ManagementPkg	ACES_Management	ErrorReq_13
ACES_ManagementPkg	ACES_Management	ErrorReq_14
ACES_ManagementPkg	ACES_Management	ErrorReq_15
ACES_ManagementPkg	ACES_Management	ErrorReq_16
ACES_ManagementPkg	ACES_Management	ErrorReq_17
ACES_ManagementPkg	ACES_Management	ErrorReq_18
ACES_ManagementPkg	ACES_Management	ErrorReq_19
ACES_ManagementPkg	ACES_Management	ErrorReq_2
ACES_ManagementPkg	ACES_Management	ErrorReq_20
ACES_ManagementPkg	ACES_Management	ErrorReq_21
ACES_ManagementPkg	ACES_Management	ErrorReq_22
ACES_ManagementPkg	ACES_Management	ErrorReq_23
ACES_ManagementPkg	ACES_Management	ErrorReq_24
ACES_ManagementPkg	ACES_Management	ErrorReq_25
ACES_ManagementPkg	ACES_Management	ErrorReq_26
ACES_ManagementPkg	ACES_Management	ErrorReq_27
ACES_ManagementPkg	ACES_Management	ErrorReq_28
ACES_ManagementPkg	ACES_Management	ErrorReq_29
ACES_ManagementPkg	ACES_Management	ErrorReq_3
ACES_ManagementPkg	ACES_Management	ErrorReq_30
ACES_ManagementPkg	ACES_Management	ErrorReq_31
ACES_ManagementPkg	ACES_Management	ErrorReq_32
ACES_ManagementPkg	ACES_Management	ErrorReq_33
ACES_ManagementPkg	ACES_Management	ErrorReq_34
ACES_ManagementPkg	ACES_Management	ErrorReq_35
ACES_ManagementPkg	ACES_Management	ErrorReq_36
ACES_ManagementPkg	ACES_Management	ErrorReq_37
ACES_ManagementPkg	ACES_Management	ErrorReq_4
ACES_ManagementPkg	ACES_Management	ErrorReq_5
ACES_ManagementPkg	ACES_Management	ErrorReq_6
ACES_ManagementPkg	ACES_Management	ErrorReq_7
ACES_ManagementPkg	ACES_Management	ErrorReq_8
ACES_ManagementPkg	ACES_Management	ErrorReq_9
ACES_ManagementPkg	ACES_Management	FuncReq_0
ACES_ManagementPkg	ACES_Management	FuncReq_1
ACES_ManagementPkg	ACES_Management	FuncReq_10
ACES_ManagementPkg	ACES_Management	FuncReq_11
ACES_ManagementPkg	ACES_Management	FuncReq_12
ACES_ManagementPkg	ACES_Management	FuncReq_13
ACES_ManagementPkg	ACES_Management	FuncReq_15
ACES_ManagementPkg	ACES_Management	FuncReq_16
ACES_ManagementPkg	ACES_Management	FuncReq_17
ACES_ManagementPkg	ACES_Management	FuncReq_18
ACES_ManagementPkg	ACES_Management	FuncReq_19
ACES_ManagementPkg	ACES_Management	FuncReq_2
ACES_ManagementPkg	ACES_Management	FuncReq_20
ACES_ManagementPkg	ACES_Management	FuncReq_21
ACES_ManagementPkg	ACES_Management	FuncReq_22
ACES_ManagementPkg	ACES_Management	FuncReq_23
ACES_ManagementPkg	ACES_Management	FuncReq_24
ACES_ManagementPkg	ACES_Management	FuncReq_25
ACES_ManagementPkg	ACES_Management	FuncReq_26
ACES_ManagementPkg	ACES_Management	FuncReq_27
ACES_ManagementPkg	ACES_Management	FuncReq_28
ACES_ManagementPkg	ACES_Management	FuncReq_29
ACES_ManagementPkg	ACES_Management	FuncReq_3
ACES_ManagementPkg	ACES_Management	FuncReq_30
ACES_ManagementPkg	ACES_Management	FuncReq_31
ACES_ManagementPkg	ACES_Management	FuncReq_32

ACES_ManagementPkg	ACES_Management	FuncReq_33
ACES_ManagementPkg	ACES_Management	FuncReq_34
ACES_ManagementPkg	ACES_Management	FuncReq_35
ACES_ManagementPkg	ACES_Management	FuncReq_36
ACES_ManagementPkg	ACES_Management	FuncReq_37
ACES_ManagementPkg	ACES_Management	FuncReq_38
ACES_ManagementPkg	ACES_Management	FuncReq_39
ACES_ManagementPkg	ACES_Management	FuncReq_4
ACES_ManagementPkg	ACES_Management	FuncReq_40
ACES_ManagementPkg	ACES_Management	FuncReq_5
ACES_ManagementPkg	ACES_Management	FuncReq_6
ACES_ManagementPkg	ACES_Management	FuncReq_7
ACES_ManagementPkg	ACES_Management	FuncReq_8
ACES_ManagementPkg	ACES_Management	FuncReq_9
ACES_ManagementPkg	ACES_Management	InterfaceReq_0
ACES_ManagementPkg	ACES_Management	InterfaceReq_1
ACES_ManagementPkg	ACES_Management	InterfaceReq_10
ACES_ManagementPkg	ACES_Management	InterfaceReq_11
ACES_ManagementPkg	ACES_Management	InterfaceReq_12
ACES_ManagementPkg	ACES_Management	InterfaceReq_13
ACES_ManagementPkg	ACES_Management	InterfaceReq_14
ACES_ManagementPkg	ACES_Management	InterfaceReq_15
ACES_ManagementPkg	ACES_Management	InterfaceReq_16
ACES_ManagementPkg	ACES_Management	InterfaceReq_17
ACES_ManagementPkg	ACES_Management	InterfaceReq_18
ACES_ManagementPkg	ACES_Management	InterfaceReq_19
ACES_ManagementPkg	ACES_Management	InterfaceReq_2
ACES_ManagementPkg	ACES_Management	InterfaceReq_20
ACES_ManagementPkg	ACES_Management	InterfaceReq_21
ACES_ManagementPkg	ACES_Management	InterfaceReq_22
ACES_ManagementPkg	ACES_Management	InterfaceReq_23
ACES_ManagementPkg	ACES_Management	InterfaceReq_24
ACES_ManagementPkg	ACES_Management	InterfaceReq_25
ACES_ManagementPkg	ACES_Management	InterfaceReq_26
ACES_ManagementPkg	ACES_Management	InterfaceReq_27
ACES_ManagementPkg	ACES_Management	InterfaceReq_28
ACES_ManagementPkg	ACES_Management	InterfaceReq_29
ACES_ManagementPkg	ACES_Management	InterfaceReq_3
ACES_ManagementPkg	ACES_Management	InterfaceReq_4
ACES_ManagementPkg	ACES_Management	InterfaceReq_5
ACES_ManagementPkg	ACES_Management	InterfaceReq_6
ACES_ManagementPkg	ACES_Management	InterfaceReq_7
ACES_ManagementPkg	ACES_Management	InterfaceReq_8
ACES_ManagementPkg	ACES_Management	InterfaceReq_9
ACES_ManagementPkg	ACES_Management	OtherReq_0
ACES_ManagementPkg	ACES_Management	OtherReq_1
ACES_ManagementPkg	ACES_Management	SafetyReq_001
ACES_ManagementPkg	ACES_Management	SafetyReq_002
ACES_ManagementPkg	ACES_Management	SafetyReq_003
ACES_ManagementPkg	ACES_Management	SafetyReq_004
ACES_ManagementPkg	ACES_Management	SafetyReq_005
ACES_ManagementPkg	ACES_Management	SafetyReq_390197
ACES_ManagementPkg	ACES_Management	Safety_Req_390198
ACES_ManagementPkg	ACES_Management	Safety_Req_390199
ACES_ManagementPkg	ACES_Management	Safety_Req_390200
ACES_ManagementPkg	ACES_Management	Safety_Req_390201
ACES_ManagementPkg	ACES_Management	Safety_Req_390206
ACES_ManagementPkg	ACES_Management	Safety_Req_390207
ACES_ManagementPkg	ACES_Management	Safety_Req_390208
ACES_ManagementPkg	ACES_Management	Safety_Req_390209
ACES_ManagementPkg	ACES_Management	Safety_Req_390210
ACES_ManagementPkg	ACES_Management	Safety_Req_390212
ACES_ManagementPkg	ACES_Management	Safety_Req_390213
ACES_ManagementPkg	ACES_Management	Safety_Req_390214

ACES_ManagementPkg	ACES_Management	Safety_Req_390215
ACES_ManagementPkg	ACES_Management	Safety_Req_390216
ACES_ManagementPkg	ACES_Management	StartUpReq_1
ACES_ManagementPkg	ACES_Management	StartUpReq_2
ACES_ManagementPkg	ACES_Management	StartUpReq_3
ACES_ManagementPkg	ACES_Management	StartUpReq_4
ACES_ManagementPkg	ACES_Management	StartUpReq_5
ACES_ManagementPkg	ACES_Management	StartUpReq_6
ACES_PowerPkg		
ACES_PowerPkg	ACES_Power	DerIntReq_15
ACES_PowerPkg	ACES_Power	DerIntReq_16
ACES_PowerPkg	ACES_Power	DerIntReq_17
ACES_PowerPkg	ACES_Power	Safety_Req_390209
ACES_PowerPkg	ACES_Power	DerIntReq_15
ACES_PowerPkg	ACES_Power	DerIntReq_16
ACES_PowerPkg	ACES_Power	DerIntReq_17
ACES_PowerPkg	ACES_Power	Safety_Req_390209

Table 5: Subsystem Requirement Allocation Table (Complete)

14 References

[1] **OMG SysML Specification 1.4** June 2015
http://sysml.org/sysml-specifications/

[2] Bruce Powel Douglass, **Agile Systems Engineering** (Morgan Kaufmann Press, 2015)
https://www.amazon.com/Agile-Systems-Engineering-Bruce-Douglass-ebook/dp/B015XPGTNI/ref=sr_1_1?ie=UTF8&qid=1478791883&sr=8-1&keywords=agile+systems

[3] Bruce Powel Douglass, **Real-Time Agility** (Addison-Wesley Professional, 2009)
https://www.amazon.com/Real-Time-Agility-Harmony-Embedded-Development/dp/0321545494/ref=sr_1_3?ie=UTF8&qid=1478791964&sr=8-3&keywords=agile+real-time

[4] Bruce Powel Douglass, **Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems** (Addison-Wesley Professional, 2002)
https://www.amazon.com/Real-Time-Design-Patterns-Scalable-Architecture/dp/0201699567/ref=sr_1_1?ie=UTF8&qid=1478792052&sr=8-1&keywords=real-time+design+patterns

[5] Hans-Peter Hoffmann, **Harmony Deskbook 4.1** (IBM, July 2013)
https://www.ibm.com/developerworks/community/blogs/35dfcb99-111b-423a-aaa4-50f3fddae141/entry/harmony_Deskbook_4_1_is_here?lang=en

[6] **PID For Dummies**
http://www.csimn.com/CSI_pages/PIDforDummies.html

[7] Bruce Powel Douglass, *Harmony MBSE Modeling Guidelines*
http://merlinscave.info/Merlins_Cave/Tutorials/Entries/2017/5/26_Harmony_Modeling_Guidelines.html