Automated Test Generation Techniques for Systems Engineering Tools

Group members: (DP46):
Imad Dodin (260713381, imad.dodin@mail.mcgill.ca),
Muhammad Huzaifa Elahi (260726386, muhammad.h.elahi@mail.mcgill.ca),
Nada Marawan (260720514, nada.marwan@mail.mcgill.ca)
Ahmed Elehwany (260707540, ahmed.elehwany@mail.mcgill.ca)

All group members are enrolled in ECSE 458-D1 & ECSE 458-D2

Project Supervisor: Daniel Varro

Abstract:

Systems Engineering tools validation is a critical but significantly costly and lengthy process in Systems Engineering. While human efforts may certify these tools to a reasonable degree of confidence, automated methods for providing test-cases to these tools are desirable to support existing efforts in tool certification. We explore the benefits of using model generation techniques with the VIATRA Generator to generate such test-cases, and lay out a plan to demonstrate how these inputs may be used to identify missing validation checks and implicit tool constraints.

Acknowledgments:

We would like to thank the following individuals, without whom's efforts, this project would not be possible:

- Professor Daniel Varro
- Mr. Aren Babikian
- Dr. Oszkar Semerath
- Dr. Gabor Szarnyas
- The VIATRA Development Team

Table of Contents

I. Introduction / Motivation / Objective

As the complexity associated with designing and managing systems increases and our systems themselves become Systems of Systems that are both deep and interdisciplinary, engineers must develop our approach to their engineering. One novel approach leverages Model-Driven Engineering approaches in the context of complex system design - this approach is called Model Based Systems Engineering and was popularized in 2007 by INCOSE (International Council on Systems Engineering) with their MBSE initiative.

While traditionally models have been a means of selective representation to describe systems and facilitate inter-team communication, Model Driven Engineering techniques systematically use models and model transformations (mapping one model to another, or to text) to increase systems' levels of abstraction and automate the development process. The process of automation, as well as the use of standardized representations is especially useful in the context of Systems Engineering where efficient means of applying changes throughout a complex system and the interdisciplinary communication of these systems are a major concern.

Models themselves are defined using models. These *meta-models* define our modelling language through a set of constraints that any given instance model of our modelling language must follow. The modelling community standard is to use EMF metamodelling language (which is based on Eclipse framework technologies) to describe such metamodels. These constraints may then be used by model generators - tools that use various techniques to automatically generate instance models that conform to language constraints (and often, additional constraints defined by the user). The leading generator with regards to scalability and efficiency, is the VIATRA Generator, which we describe in greater depth in the Background section of the report.

MBSE tools like Capella (which itself uses the Arcadia method for Systems Engineering), provide well defined models and layers of abstraction that are often used to define complex aerospace systems. Tools such as these must be qualified for use in safety-critical systems, and while manual certification efforts certainly dive deep into the tools inner workings, they are extremely expensive, both in financial cost and time. It is desirable to at least partially automate the qualification process of such tools, both to increase efficiency and completeness.

The aim of this project is two-fold: First, we aim to develop a system model using the Capella framework - this is important both for our own exposure to current trends in the industry and to address the current lack of availability of such models to the public. Secondly, we aim to use the VIATRA Model Generator to generate instance models that we may input into Model Based Systems Engineering tools in order to systematically verify them.

II. Background

Modelling

This project report involves various references to core concepts of modelling, that are widely used in MBSE. Modelling tools such as Capella have their own defined syntax, which allows the user to create valid models by following that syntax. In other words, a modelling language defines rules preventing model violations such as missing associations, circular dependencies and model conflicts. That way the modelling language eliminates a huge subset of modelling inaccuracies and produces more reliable models for engineering systems. Modelling tools including Capella are used intensively within the Arcadia methodology.

A model uses different types of diagrams to depict different aspects of the system. In the context of Capella language, these diagram types are referred to as “layers”. It’s crucial to decide on a scope for the model, which defines the functionalities and components to be included in the modelled subsystem. It’s also important to be familiar with the modelling terminology such as actors(entities) and use cases(features).

Eclipse Plugin Development

This project requires core knowledge of Eclipse plugin development, since most of the tools involved are used in the form of Eclipse plugins. A plug-in[8] in Eclipse is a component that provides a certain type of service within the context of the Eclipse workbench. Eclipse Modelling Framework (EMF) provides a set of Eclipse plug-ins which provide facility for modelling and code generation. The Viatra tool used for code generation of instance models is also provided in the form of an eclipse plug-in.

Testing

Modelling certified tools are manually tested to make sure they accurately fulfill the modelling requirements so that the output conforms to certain use cases. This type of certification is feasible for limited-scale development of models. On large scale models, however, it becomes very difficult to manually verify the model due to the many possible degrees of freedom involved and the low feasibility in terms of resources and time to manually provide all the possible large-scale edge cases and scenarios.

Model generation tools today are capable of providing automated means of generating very detailed and large-scale models. Consequently, these code generation tools enhanced the testing process for modelling tools by providing more timesaving and practical ways of validating large-scale scenarios and cases. The testing process now can be enhanced by using the generation tool to output a large set of models of various sizes and edge cases, and plug in these models into the modelling tool. Observing the behavior of the modelling tool would therefore give insight into the performance and specifications of the tool such as the size of models to be handled by the tool, the set of edge cases allowed and possible missing constraints.

Eclipse Modelling Framework (EMF)

EMF[9] is a modeling framework and code generation facility for building tools and other applications based on a structured data model. For models, it provides Java interfaces and implementation classes for all the classes in the model, in addition to a factory and package implementation class. EMF can be used to define models on Eclipse. EMF could also be used to create a model defining a modelling language. This model is called a metamodel (model of a model).

In the context of this project, EMF is a core requirement for all the tools involved. Code generation of instance models requires us to load the corresponding metamodel of the modelling software for which we are generating the instance models. This metamodel defines the syntax and constraints of the modelling software. Therefore, loading the metamodel to the code generation tool allows us to generate instance models that conform to the modelling software.

Capella

Capella[10] is an open source, comprehensive, extensible and field-proven MBSE tool to successfully design system, software, and hardware architecture. Relying on Arcadia methodology, Capella helps systems architects to formalize specification and master architectural design. Capella models consist of several layers, following a top-bottom approach, where the top layers define basic requirements of the system, then as we progress into deeper layers, lower level implementation details are provided.

YAKINDU

YAKINDU[11] Statechart Tools is a modular toolkit for developing, simulating, and generating executable finite-state machines (FSM) or simply state machines. Initially, it has been designed for the embedded systems industry: automotive, system controls, vending machines etc. However, this toolkit is bringing the benefits of finite-state machines (FSM) and Harel statecharts to everyone who needs to design, simulate and develop behavior. Since Yakindu is based on the Eclipse platform, you can create Yakindu models simply by installing the Yakindu plug-in on eclipse. You can also download the Yakindu software and create models independently from eclipse.

VIATRA

VIATRA[12] Solver is a framework for automatically generating well-formed instance models for domain-specific languages. The framework natively supports instance generation for EMF metamodels, optionally, by extending an existing model. The generated models are presented as standard instances of the target domain. The solver creates models that satisfy well-formedness constraints of the target language, which is given as an input to the framework. The framework translates the language specification to a logic problem, which is solved by an underlying first order logic solver to create valid graph models. The solver has a direct dependency on EMF to be able to load the metamodel corresponding to the language specification.

III. Requirements and Problem

Systems Modelling with the Arcadia Method

In order to investigate Systems Modelling, we decided to create an end to end model using the Capella software mentioned above. Some of the questions we wished to answer were:

a) Understand the objectives of the Arcadia Method that Capella implements?
b) Understand how an encapsulated system is decomposed in the various abstraction layers Capella supports?

Arcadia[1] itself is a system engineering method to system requirements by dividing a system into interlinked layers. Capella is a manifestation of this approach in a condensed software. To investigate Capella to the fullest extent, we decided to create a model of a relevant system of our choosing to understand how to formula an end to end system.

Capella supports five layers for modelling:

1. Operational Analysis
2. System Analysis
3. Logical Architecture
4. Physical Architecture
5. EBPS

Operational Analysis

Operational analysis describes the high level relationship between stakeholders, actors and components while determining what users of a system must accomplish without delving into any implementation details.

System Analysis

System analysis goes into system information and overview and models the flow of the system and each actor's actions or responsibilities. It comprehensively describes high level functionality of the system but again without implementation specifics. This layer describes data exchanges throughout the system by virtue of the interactions and capabilities already specified in the operational analysis layer.

Logical Architecture

Logical architecture defines how the system operates and further refines the functions we defined in system analysis. Functions are broken down further to increase granularity at this level of abstraction and allow refined functional analysis. We can start making components as well as creating subcomponents for logic that exist within a larger component. Logical architecture is still independent from physical implementation. It is possible to derive various different physical implementations despite commonalities in modelling at this layer and it's level of abstraction.

Physical Architecture

Physical architecture breaks down the divided components of the layers above into their functional compositions, highlighting exactly how to realize the previously defined capabilities and functions. Instead of defining at a higher level, implementation details are more readily seen in white box form.

EBPS

A component breakdown into granular subcomponents to understand the hardware required to realize the system modelled with the previous four layers of architecture and requirement design

Traceability and Transitions

The above layers are linked together using traceability links that transcend layers. This means a capability created in operational analysis is linked to a system function and a logical function at layers below. These links ensure the model is kept concise and edits made at any layer correspond to changes throughout the model for relevant impacted models.

Actors and functions defined in a layer can automatically “transition” from a higher layer to a lower one, allowing us to have new copies of the actors and capabilities that are linked automatically to the ones at a higher level. We can then drop and generate new models at the current layer and add more functions and actors that are not necessarily reflected at higher levels but should be transitioned downwards throughout the model

Using these features of Capella we designed our own model, the results of which are highlighted in the Results section below.

Model Generation with VIATRA Generator

As described in the Introduction section, we aim to leverage the VIATRA model generator technology to generate test input into our tool. Our end goal is to achieve two high level goals by these means:

a) Discover erroneous input cases into our tool. I.e. How large of a model can our tool handle? Verify it can correctly achieve its intended functions over a suite of large and diverse test inputs?
b) Discover any additional constraints that are implicitly enforced by the Capella tool. I.e. While our metamodel defines a set of constraints, the tools functionality may rely on another implicit set of constraints - we aim to demonstrate that systematic use of model generation techniques may discover such cases fairly inexpensively.

Our initial end goal was to generate such inputs for the Capella tool, however, after an initial investigatory period, we noted a significant amount of complexities associated with generating instance models for the Capella tool. This complexity is associated with the use of several technologies that post-process Instance Models for Capella (i.e. additional transformations must be performed before an instance model can be input into Capella). As a result, we have decided to introduce a new end goal to demonstrate the use of model generation on Yakindu Statecharts, a tool with less such complexity to allow us to focus our main efforts and time budget on achieving our proof of concept for the project.

While this requirement seems simple enough, it is not easy to handle models defined by a third-party, nor is it easy to set up the required VIATRA Generator and its dependencies. Indeed, these were major deliverables on their own:

I. Identify current conflicts between the Eclipse Modelling Framework and VIATRA Generator - in our attempts to set-up the generator, we made a note of several versions of the Generator tool and the Eclipse Modelling Framework that were not functional and reported these to the development team.
II. Identify required post-processing on generated instance models before they may be input into the tool themselves. A common case is that models define their abstract syntax (conceptual components) and concrete syntax (diagrammatic representations) separately. As such, the model generator will output a model that is conceptual complete, but with no associated diagrammatic representation information - which is obviously a requirement to be input into the associated tool. As an example, in the case of Yakindu, we aim to use Choco, a Constraint Programming library to help us define component coordinates when generating the diagrammatic representation of our instance model out of the conceptual components output by the generator.

In the case of Capella, the latter is a major concern as Capella is a tool that is significantly more complex and appears to have a great length of post-processing that is not publicly documented. We note however that, although Capella is an industry-grade Systems Engineering tool and, as such, it would be valuable to use it to demonstrate our proof of concept, demonstrating on Yakindu is also sufficient in showing the value of model generation techniques in tool qualification, demonstrating the use of generated instance models to find errors in a systems engineering tool.

IV. Design and Result

In terms of designing a comprehensive solution for our stated problems and objectives, we divided tasks along two relatively binary paths - Capella modelling and instance model generation.

Capella Modelling

On the modelling front, work was done to delve into the details of Capella. Extensive research was done to first understand why Capella can be used to model extensive systems and understand the Arcadia method that Capella implements.

Arcadia itself is a system engineering method based on the use of models, with a focus on the collaborative definition, evaluation and exploitation of its architecture. Arcadia is designed[1] as a tool to define user and system needs, design and validate system requirements and flesh out software and hardware architectures through a multi layered approach.

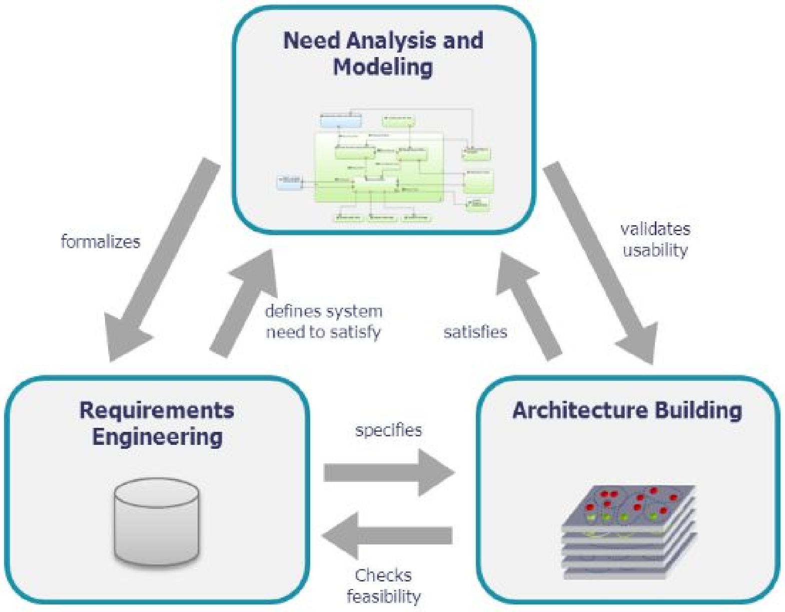


Figure IV-1: Arcadia[2] promotes a viewpoint-driven approach and emphasizes a clear distinction between need and solution.

The Arcadia method itself was developed by Thales Group[3] between 2005 and 2010 through an iterative process involving operational architects. Thales Group is a French multinational company that designs and builds electrical systems and provides services for the aerospace, defence, transportation and security markets.

To understand how Capella implements this method we decided to model a system of our choosing: an AFCS[4] - Aircraft Flight Control System. The AFCS system allows us to investigate a use case problem for which Capella is actually used[5] in the Aerospace industry.

The AFCS shall provide numerous functionalities[6] including high precision flight path control with precise lateral and vertical accuracy, stabilization of the aircraft attitude in atmospheric turbulence and automatic landing.

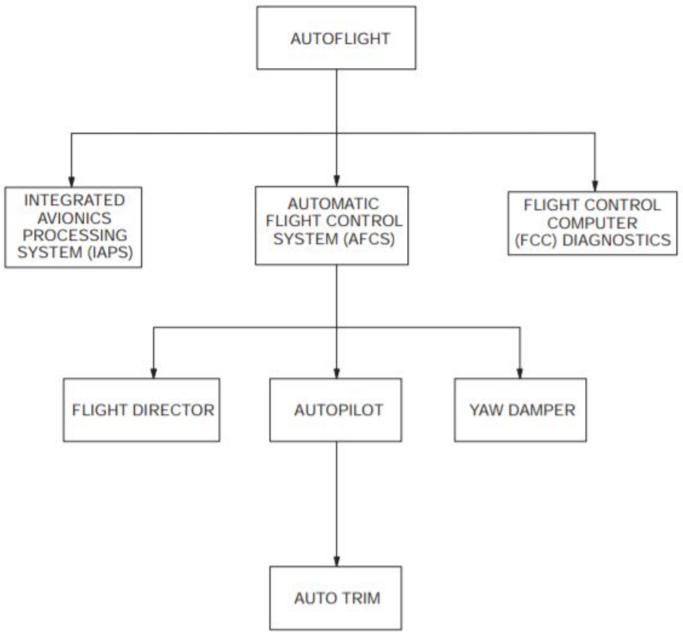

AUTOFLIGHT
INTEGRATED AVIONICS PROCESSING SYSTEM (IAPS)
AUTOMATIC FLIGHT CONTROL SYSTEM (AFCS)
FLIGHT CONTROL COMPUTER (FCC) DIAGNOSTICS
FLIGHT DIRECTOR
AUTOPILOT
YAW DAMPER
AUTO TRIM

Figure IV-2: AFCS Architecture[4]

Capella consists of **five** layers[7] of abstraction when it comes to modelling layers. In the process of our investigation these past two semesters, we managed to proceed to the fourth layer using a top down approach, going from a more abstract layer to a more granular layer with each iteration. The four layers investigated were the **Operational Analysis**, **System Analysis**, **Logical Architecture, Physical Architecture** layers. The details for the following sections were taken through referencing *System Architecture Modelling with the Arcadia Method - a Practical Guide to Capella* by Pascal Roques.

Note: All of our models can be found in the appendix section 1.

Operational Analysis

We created capabilities for the functionalities that the AFCS is meant to fulfil. The actors and entities associated within the scope of the system were assigned and capabilities were allocated to these actors and entities with links between and, hierarchies of, capabilities. In general, capabilities were grouped under “Set Flight Path”, “Control Navigation” and “Configure Autopilot”.

The main actors and entities were:

- The Air Data System which records flight data
- The Safety System which monitors sensor readings and sends signals to interrupt the autopilot and issue warnings.
- The Flight Director which provides navigational assistance to follow a determined path
- The Autopilot which automatically takes control of flight navigation and guides an aircraft from one location to another and can perform a landing
- The AFCS itself which is a generalization of the Autopilot and Flight Director

Below is one of the models created to show the architectural design involving actors/entities and their capabilities.

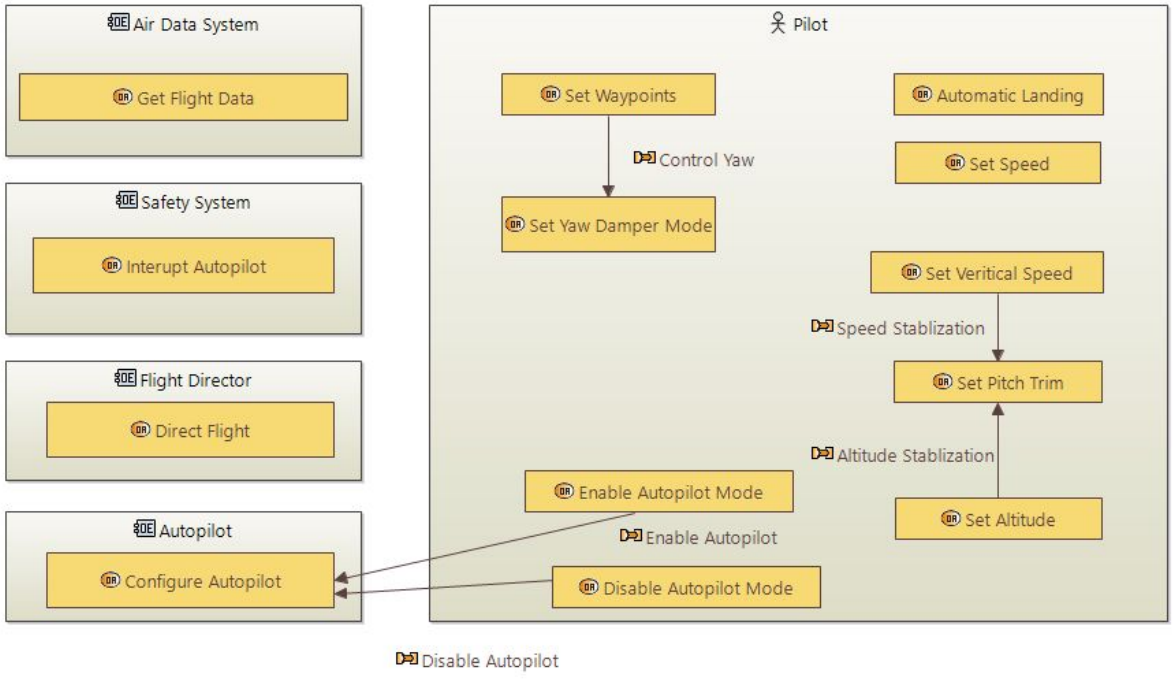


Figure IV-3: OAB Diagram - Operational Architecture

System Analysis

The abstraction of this layer can be described as similar to calling functions (modelled as system function) and showing which calls occur (modelled as functional exchanges) through a triggered call but keeping the functions themselves as a black box. Models similar to sequence diagrams known as scenarios allow greater granular details in terms where and when functional calls are made. These functional exchanges are labelled with the functional element being passed from one function to another through the call, i.e useful parameters needed for function B are passed through function A when function B is called from function A.

Furthermore, since functions are divided and only linked by a one way functional exchange with one defined input and one defined output, we can create a chain of linked functions. If a chain of such functions represents a logical composition comprising a common use case instance, we can create a formal functional chain. This chain allows us to clearly see inputs and outputs for only the relevant selected functions to perform an operation or specific use case instance.

It is logical to have a functional exchange between components go through the system itself rather than have various components interacting together completely independent of the system and that is how we modelled our AFCS. The system performs various system functions to facilitate implementing the capabilities defined in the Operational Analysis layer by connecting various functions assigned to entities and actors. The following diagram shows the System Architecture:

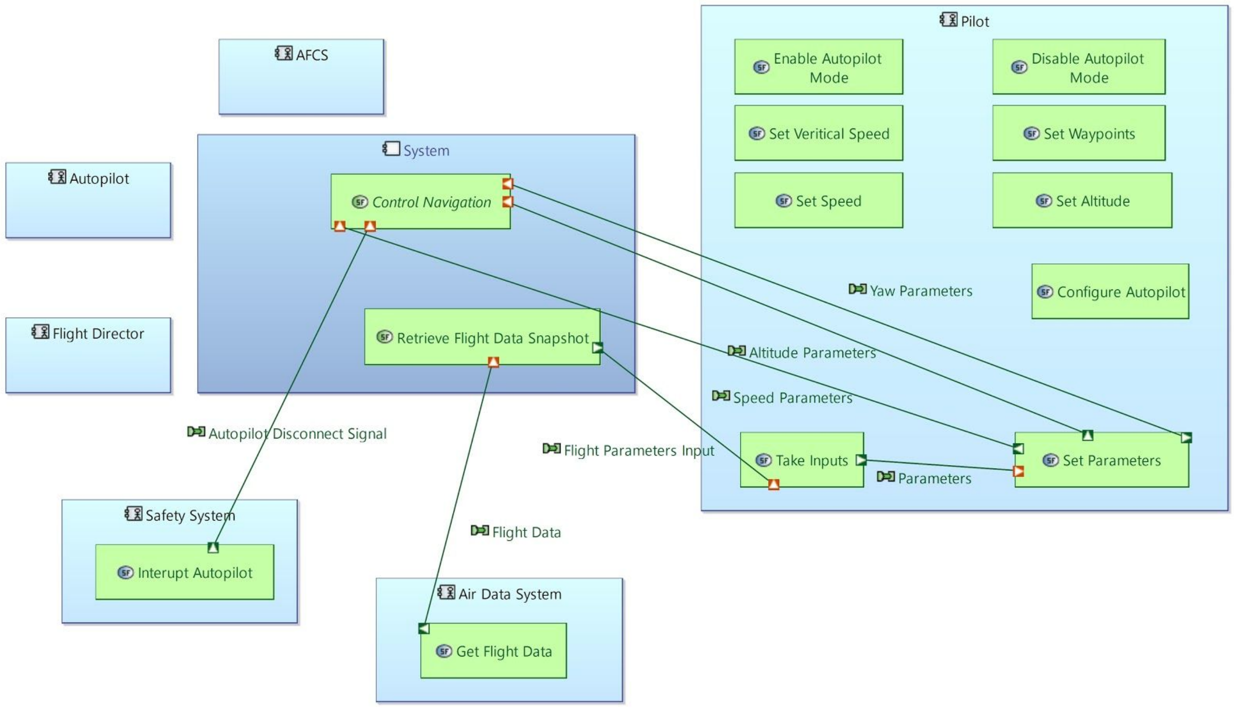


Figure IV-4: SAB Diagram: System Architecture

Logical Architecture

Similar to the System Analysis layer, the Logical Architecture layer shows functions and functional exchanges of data modelled throughout the system, however in greater granular detail. Functions treated as black boxes can be exposed further in this layer with underlying subfunctions that comprise the larger function developed in high level layers.

For instance we decided to elaborate upon various subsystems. These included the navigational subsystem, the safety subsystem and the data subsystem. We expanded on the logical functions within these subsystems by elaborating on feedback loops. Each subsystem first acquires relevant data then proceeds to send the data to a detector/parser then continuously uses accumulated data to validate the readings. Using this feedback loop to ensure that data propagation is correct and any outliers don’t cause a regression in behavior and only sustained patterns of data collection make an impact. This reflects the high volume of data collection that takes place in such an aircraft system and allows us to sample a larger set of readings. The following architectural diagram contextually describes the design of the layer as elaborated upon here:

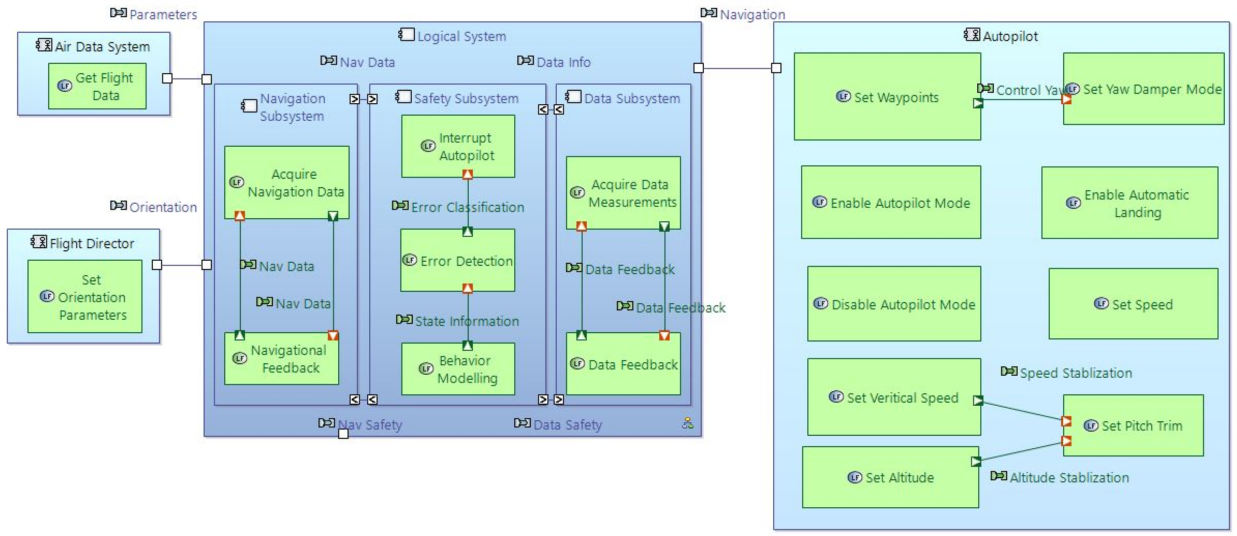


Figure IV-5: LAB Diagram: Logical Architecture

Once again, we were able to harness Capella's transitions from higher levels to automatically gain access to pre-established actors and systems from the Operational & System Analysis layers described earlier.

Physical Architecture

The physical architecture layer strips away abstraction as much as possible to unveil implementation details and specifications. These details manifest themselves as outlines of individual components. For example, we only described navigation first as an operation very abstractly. Then as we went further through layers of Capella we defined it with the relevant actors and associated functions that enable the navigation to take place.

At this physical layer we are now able to show the physical components that make up the design of the system. We for example defined a sensor holder comprising all the various sensors required for the system and subdivided the holder by relevant sensor types such as navigation, temperature.

We also added flight control components such as the rudder, aileron and elevator that combine to provide 3-dimensional control to an aircraft. Each navigational sensor again has a feedback loop for the reasons provided in the layer above. The physical architecture can be viewed below in figure IV-6:

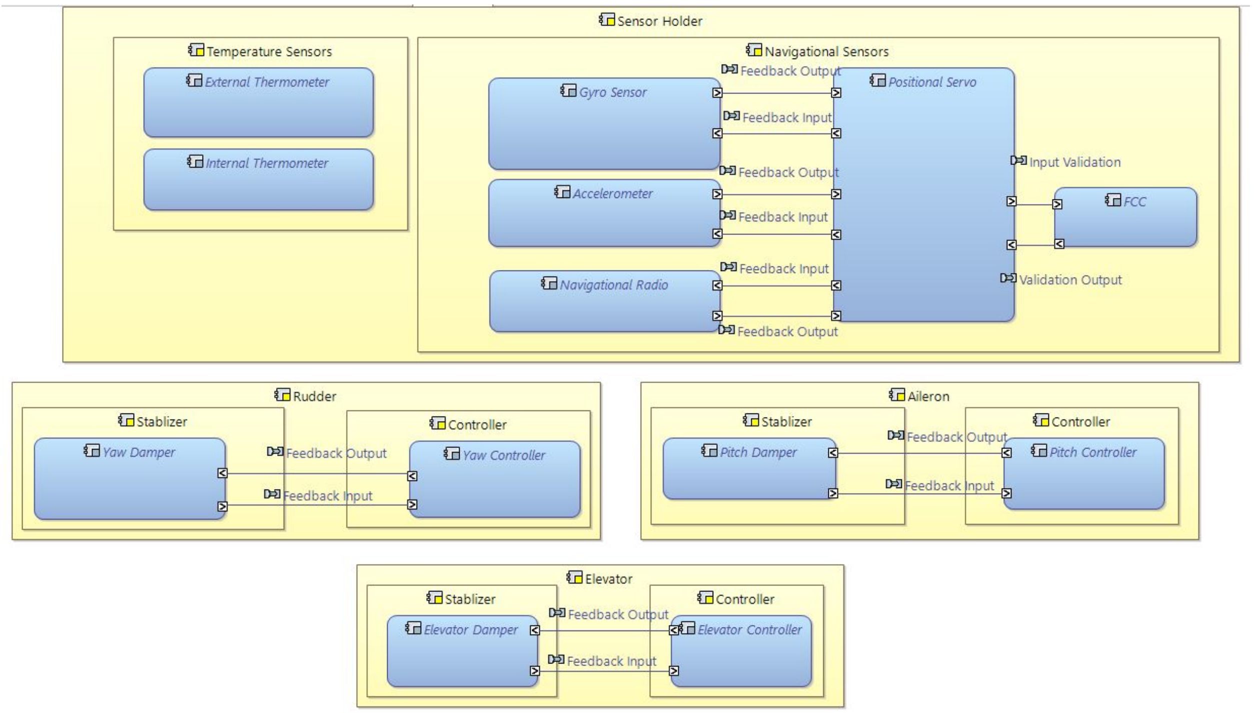


Figure IV-6: PAB Diagram: Physical Architecture

Plugins

The M2Doc plugin was used to automatically generate text documents outlining the structure of our project and its various models. We used pre-rendered templates to generate two documents, a system level document and a logical function document. Due to time constraints we were unable to delve into the templates to edit them ourselves but they provided valuable insight into how Capella models can be exported in different formats. Tables and diagrams provided context for how different entities were related throughout the layers. The more the level of detail invested into the models themselves, i.e descriptions for models, attributes for entities, the higher the corresponding detail in the generated documents. The documents enforced no strict rules on the model and were flexible enough to null out any irrelevant fields and posed no blockages to our model development itself.

Instance Model Generation

Simultaneously to the development of realistic Capella models, we put significant efforts into automating the generation of instance models for different modelling software tools. A major product of our efforts was the generation of instance models for Yakindu Statechart Tools to demonstrate the ability to employ model generation in the context of testing Systems Engineering tools.

Yakindu as a Proof of Concept

While initially we aimed to generate instance models for Capella, in conformance with the modelling efforts outlined in the preceding section, a significant decision was instead to focus efforts on Yakindu Statechart Tools due to non-trivial complexities associated with post-processing generated Capella models. In the interest of time, we employ Yakindu Statechart Tools, a tool with less associated complexities which allows users to design and execute Statechart definitions, as a proof of concept for the use of model generation techniques in testing systems engineer tools.

Model Generation Using VIATRA

Execution Procedure:

1. Extract metamodel for a language specification
2. Load the Metamodel.
3. Define well-formedness constraints for generation.
4. The framework transforms metamodel into a logic problem.
5. The VIATRA Reasoner solves the logic problem and outputs a solution that conforms to both metamodel and user-defined WF constraints.
6. The framework transforms the solution into an instance model.

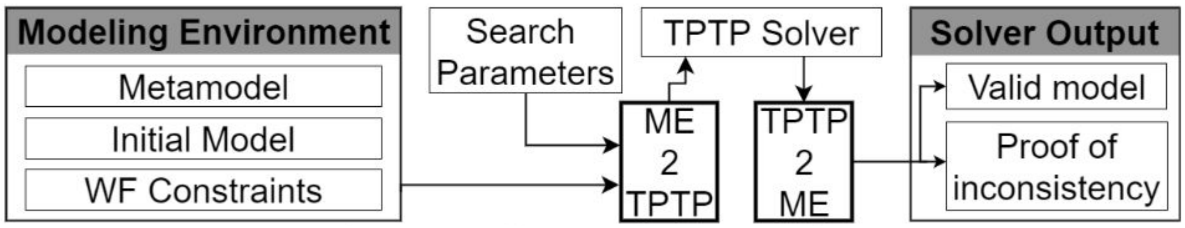


Figure IV-7: Overview of VIATRA Model Generation Approach

As outlined above, the first step in the process is to extract the metamodel file(s) that specify the modelling language for which we wish to generate instances. This is sometimes not trivial during research, depending on whether such models are made publicly available, however, we do not consider this a concern in practice where tool qualifiers have such definitions made available to them.

VIATRA Model Generator is, at its core, a logic solver - i.e. a program that solves a problem given a set of constraints. In this light, the bulk of the efforts in the generation process lie in specifying said constraints. We consider the extracted metamodel discussed before as a set of constraints for a valid instance of the specified language and may optionally reinforce this metamodel with additional constraints in the form of (i) an initial model and (ii) well-formedness constraints, the former constraining the generated instance to include all elements of the initial model.

We draw attention, then, to the additional well-formedness constraints and note the importance of these constraints in the context of systematically testing MBSE tools. These user-defined constraints can be used to fine-tune the output to more specific generated outputs based on the functionality of the tool which we are testing. As an example from our Yakindu proof of concept, to systematically test whether or not synchronization states are correctly handled by the tool, we can constraint that such states must be generated by the generator, or even, the amount of such states that are generated and the amount of transitions into said states.

Once constraints are input into the generator, the generator maps these constraints to a Logic Problem which it then solves generating either a set of instance models conforming to the input constraints or a proof of inconsistency in the event that the input constraints are inconsistent (e.g. conflicting) amongst one another. The generated instances often cannot be directly input into a tool, with a certain degree of post-processing required before we can do so. We discuss this in the context of Yakindu in the section that follows.

We note that, omitted from above, are the significant efforts associated with the set up of the actual modelling environment and generator which, as with many products of research, is a finicky and time consuming process due to conflicts in various versions of the generator's dependencies (which we also note are reported to the development team). We discussed these efforts previously in Section III.

Post Processing

Generated instances often have a certain degree of post-processing required before we can input them into their associated tools. We identify and explore 3 required steps in post-processing for the case of Yakindu:

I. Addition of name tags.
II. Generation of concrete elements.
III. Generation of concrete syntax physical bounds (Choco Solver)

(I.) The Yakindu tool expects each element to have a name associated with it. The generated models lack name tags, since the Viatra generator does not support attribute generation. Therefore, we had to programmatically generate name tags for elements as part of our post-processing.

(II.) Yakindu models are composed of two sections: abstract syntax and concrete syntax. The abstract syntax defines the conceptual components in the model and the concrete syntax defines the diagrammatic representation for the model components. The metamodel for Yakindu Statechart Tools covers only the language's abstract syntax and thus the model generator outputs a model defining the abstract syntax, but with no associated diagrammatic representation information (concrete syntax). As a result, post-processing is required on generated instance models before they may be imported into the Yakindu Statecharts Tool.

Yakindu Statechart Tools uses a third-party tool for diagrammatic representation, known as the Graphical Modelling Framework (GMF). GMF is a general-use framework with its own defined notation for graphical and concrete representation. We were able to extract the metamodel for GMF language and generate its model code to use it later for generation of concrete elements to support diagrammatic representation. After extensive reverse engineering and careful analysis of real Yakindu models, we were able to converge on an abridged set of concrete elements that are required by the Yakindu modelling tool. The generation of a concrete syntax was implemented programmatically by traversing the abstract model and recursively generating concrete elements corresponding to each abstract element and associating each said concrete element to the abstract element which it represents, as required by GMF. We note that we generate only the required concrete elements for the tool to handle the model and not necessarily to visualize it. The model in Appendix 2.1.1 shows that, consequently, elements are not always rendered correctly by the tool. The extension to generate elements required for rendering is trivial but time intensive and was omitted in this design project in the interest of time.

(III.) GMF requires physical bounds to be defined for each of the concrete elements that it defines, using a cartesian coordinate system. To generate these bounds, we used a constraints programming solver library called Choco which takes as input constraints (i.e. that two elements must be contained in any encompassing element and cannot overlap) and outputs valid coordinates satisfying these constraints.

We provide Figure IV-8 for a visual summary of the post processing steps required, by means of the XML representation of an instance model. We similarly provide the XMI representations of both the Model Generator output and its associated post-processed model in the Appendix, for reference.

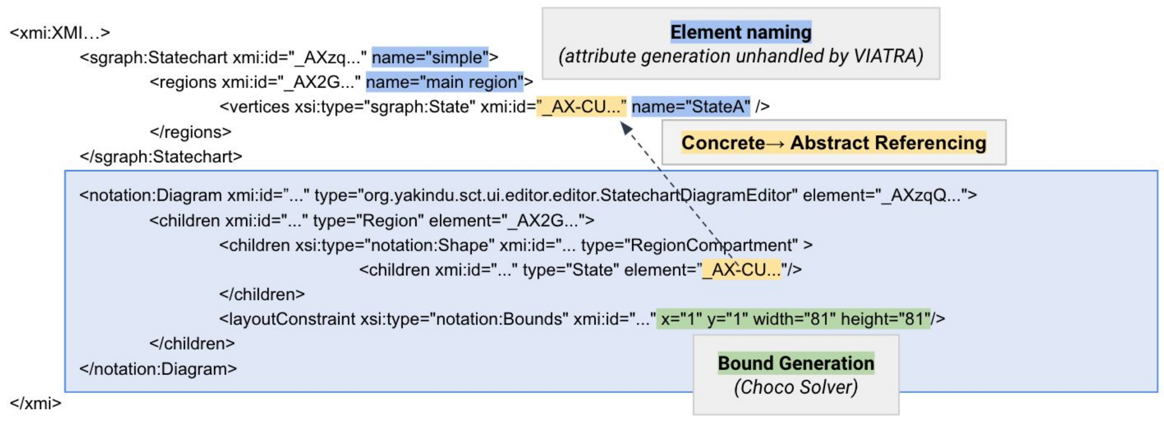

```
<xmi:XMI...>
        <sgraph:Statechart xmi:id="_AXzq..." name="simple">
                <regions xmi:id="_AX2G..." name="main region">
                        <vertices xsi:type="sgraph:State" xmi:id="_AX-CU..." name="StateA" />
                </regions>
        </sgraph:Statechart>
        <notation:Diagram xmi:id="..." type="org.yakindu.sct.ui.editor.editor.StatechartDiagramEditor" element="_AXzqQ...">
                <children xmi:id="..." type="Region" element="_AX2G...">
                        <children xsi:type="notation:Shape" xmi:id="... type="RegionCompartment" >
                                <children xmi:id="..." type="State" element="_AX-CU..."/>
                        </children>
                        <layoutConstraint xsi:type="notation:Bounds" xmi:id="..." x="1" y="1" width="81" height="81"/>
                </children>
        </notation:Diagram>
</xmi>
```
Element naming
(attribute generation unhandled by VIATRA)
Concrete→ Abstract Referencing
Bound Generation
(Choco Solver)

Figure IV-8: Overview example of post-processing, diagrammatic representation shaded in blue.

Generated Models as Test Inputs

The focus of this project is, ultimately, to use the generated output as test inputs for MBSE tool verification. We outline the approach for this and demonstrate with an example on how we can use the generated output to achieve this goal:

We note that, as a consequence of being generated by a logic solver, the generated outputs lack any semantic meaning and that this is indeed a benefit of this approach as it does not rely on any insight into how the tools will be used (a crucial property in Systems Engineering as tool qualifiers can seldom predict what will be designed by the tools they verify). Instead we focus on using these models to verify the functionality of these tools according to the constraints set forth only by the language they specify.

As a consequence of the lack of semantic meaning, we find that when output into the MBSE tool, one may find a poorly formatted, messy output. An example of this can be found in Appendix 2.1.1 where we provide a screenshot of a generated output being visualized in Yakindu Statechart Tools.

In the context of Yakindu Statechart Tools, as is typical with many other MBSE tools, a major concern is whether the tool's model verification functionality functions correctly (i.e. whether it reports errors in the inputs provided to it). In the case that no additional Well-Formedness Constraints are specified, we can simply manually inspect models and compare them with the errors reported by the tools to verify that the appropriate errors are caught. An example of these errors corresponding to the model in Appendix 2.1.1 is found in Appendix 2.1.2.

By specifying additional Well-Formedness Constraints, we have greater control on the output of the generator and can enforce that specific states be generated and thus do not need to rely on manual inspection to verify the errors reported by the tool. Indeed, by combining these constraints with the ability to generate extremely large models, we can verify such errors at a scale not feasible with manual test input construction.

V. Impact on Society and the Environment

Systems Tool Verification is a significant societal concern - particularly in the context of safety-critical systems like aircraft, health-care devices and transport vehicles, where an error can potentially result in a loss of human life. With a recent increase in concerns for aircraft safety, associated with the recent crash of two Boeing 737 MAX planes, and the models subsequent grounding by most major airlines, we highlight the particular importance of ensuring confidence in safety-critical systems today.

It is not surprising that such certification efforts are costly, both in time and financial costs. While 3rd party qualifiers do their due diligence in qualifying systems engineering tools, there are limits on the diversity of inputs that may be tested and thus on the confidence we may have in these tools. Automated generation techniques provide a low-cost means to support existing verification efforts - we suspect that when integrated into existing verification efforts, these techniques will increase the confidence in the tools they certify and therefore in the systems who depend on these tools.

We note, however, that caution must be taken when using automated techniques. A reliance on automated methods may have adverse effects on certification efforts and conversely lower confidence in the tools which they verify. Crucially, we highlight the importance that these techniques be integrated into existing efforts and not completely replace them - outputs from generated models should still be reviewed by experienced Systems Engineers.

While our project has significant impacts on the overall confidence of systems, by improving the tool certification process, it does not have a significant direct impact on the environment. As a principally software oriented development effort, there is likewise no significant direct impact on the environment as a result of our own efforts. We highlight, however, that intentional models (that encompass the intentions of primary stakeholders of our systems) may highlight environmental concerns as a principal concern during the design of the system.

VI. Report on Teamwork

Nada Marawan - With some previous experience in the Aerospace Industry and currently following a minor in Aerospace Engineering, Nada acted as a domain expert when modelling the, at times complex, Aircraft Flight Control System. We enumerate some of her high level responsibilities below:

- Develop an understanding of the Capella Systems Engineering Tool and its associated Arcadia method, within the context of aerospace systems.
- Develop four layers of models following the Arcadia method, namely: (i) Operational Analysis, (ii) System Analysis and (iii) Logical Architecture (iv) Physical Architecture.
- Ensure that the models are an accurate representation of a functional Aircraft Flight Control System.
- Provide insight into the Software Engineering field in the context of the Aviation Industry (general practices, culture etc…)
- Ensure that the model components are inline with naming conventions in the fields of software validation and fault tolerance.

Huzaifa Elahi - Huzaifa focused mainly on Capella modelling and review. His high level responsibilities were a follows:

- Develop an understanding of the Capella Systems Engineer Tool and its associated Arcadia method, within the context of aerospace systems.
- Develop four layers of models following the Arcadia method, as above: (i) Operational Analysis, (ii) System Analysis and (iii) Logical Architecture (iv) Physical Architecture.
- Discuss and document comments made by our supervisor with regards to the models, assisting in implementing changes as appropriate to be consistent with the modelling language.
- Creating a documentation report explaining in detail the different models and highlighting their main components and use.

Imad Dodin - Imad focused mainly on model generation techniques, alongside Ahmed. This was an effort that involved a significant amount of experimentation, trial and error and debugging. With some basic experience with such technologies due to a relevant past course taken, he provides context and ramp-up for concepts related to modelling and modelling technologies. We enumerate his high-level responsibilities below:

- Provide context into modelling technologies and concepts, describing the concept of modelling languages, metamodels and transformations.
- (Further) explore relevant technologies: EMF, eCore, Xtend, Viatra Query Language, Sirius and Xtext.
- Explore and experiment with relevant modelling languages: (Capella) and Yakindu.
- Limited assistance in debugging issues with installation of required tools: Viatra Generator and EMF.
- Development of required post-processing: recursive generation of concrete syntax with an abstract syntax input.
- Development of required post-processing: generation of concrete syntax physical bounds with use of the Choco Constraint Programming solver.

Ahmed Elehwany - Ahmed also focused on model generation techniques. Ramping up was non-trivial, due to the amount of contextual knowledge about modelling that was required, but was accomplished swiftly.

- Ramp up on modelling technologies and concepts (modelling languages, metamodels and model transformations).
- Explore relevant technologies: EMF, eCore, Xtend, Viatra Query Language, Sirius and XText.
- Explore and experiment with relevant modelling languages: (Capella) and Yakindu.
- Major efforts in debugging issues with installation of required tools, following up with supervisor and graduate students with extensive experience in these tools.
- Investigation of required post-processing steps: lengthy manual testing process to identify required concrete syntax elements for proper functionality of the Yakindu Tools.
- Analysis of generated test inputs on Yakindu Statechart Tools (i.e. manual inspection of output of Yakindu given generated inputs).

VII. Conclusion

The workflow of our project was split into two main scopes; creating a Capella model for Air Flight Control System(AFCS) and automated instance generation for Yakindu models.

For the former, we have successfully developed the 4 layers of the model. The layers were developed such that the model is well formed and compliant to the Capella language specification and validation rules. Research was done by the team to make sure the developed model is also consistent with the domain, such that it provides an accurate source of documentation for the modelled system. We additionally included documentation generation capabilities to automatically render documents for our model using various plug-ins that we investigated. We are satisfied that our model is a reasonably accurate representation of such an AFCS model and broadly encompasses much of the required criteria necessary for the system.

We believe realistic future objectives that could build upon our work include delving into the final Capella layer for component breakdown. Exposing the details in the final layer would provide relevant knowledge for deployment of a system and its physical requirements based on the already defined layers. Additionally generating our own templates in M2Doc for documentation of the model would provide a helpful step for ensuring system specifications are comprehensively maintained. Documentation generation could also be automated as part of a pipeline for development and a form of version control.

We similarly explored the automated model generation technology. First, we spent time ramping ourselves up and investigating other previously implemented model generation projects(i.e Gamma Project) to acquire sufficient background knowledge regarding EMF and the model generation technology. Once familiarizing ourselves with the relevant technologies and extensive debugging to set up the modelling workspace, we explored the Yakindu language in depth, discovering and extracting its concrete representation language definition. We were finally able to demonstrate a rudimentary proof of concept of using the VIATRA model generator to help automate the testing process for Yakindu as a proof of concept tool for MBSE, highlighting the associated complexities in post-processing to achieve this goal.

There are various future efforts for this project which we suggest: firstly and most trivially, we suggest the extension of the existing concrete representation generation to include elements required for correct rendering of the Yakindu models. We also suggest the exploration of leveraging our efforts here to discover implicit constraints posed by the Yakindu tool (i.e. those constraints which the tool requires but which its language definition does not). Finally, we suggest the exploration of generalizing the concrete syntax generation process for GMF - we note that various Eclipse based tools (and their associated modelling languages) leverage this tool and generalization of concrete representation for it is beneficial to the practicality of using model generation approaches. We pose the possibility of constructing a tool that simply takes in mapping rules from abstract elements to their concrete representations and generates the concrete representation based on these constraints.

References

[1] “Arcadia Datasheet,” Arcadia Datasheet. [Online]. Available: Arcadia Datasheet.

[2] “Let yourself be guided with Arcadia,” *Capella MBSE Tool - Arcadia*. [Online]. Available: https://www.polarsys.org/capella/arcadia.html. [Accessed: 28-Nov-2019].

[3] “Home Thales,” *Home Thales*. [Online]. Available: https://www.thalesgroup.com/en. [Accessed: 28-Nov-2019].

[4] “Canadair Regional Jet 100/200 - Automatic Flight Control System,” *Canadair Regional Jet 100/200 - Automatic Flight Control System*. [Online]. Available: http://www.smartcockpit.com/docs/Bombardier_CRJ_200-Automatic_Flight_Control_System.pdf.

[5] “Open Source Solution for Model-Based Systems Engineering,” *Model Based Systems Engineering | Capella MBSE Tool*. [Online]. Available: https://www.polarsys.org/capella/. [Accessed: 28-Nov-2019].

[6] L. Dalldorff, R. Luckner, and R. Reichel, “A Full-Authority Automatic Flight Control System for the Civil Airborne Utility Aircraft S15 – LAPAZ,” *A Full-Authority Automatic Flight Control System for the Civil Airborne Utility Aircraft S15 – LAPAZ*, 12-Apr-2013. [Online]. Available: https://aerospace-europe.eu/media/books/delft-0098.pdf. [Accessed: 28-Nov-2019].

[7] P. Roques, *Systems Architecture Modeling with the Arcadia Method: A Practical Guide to Capella*. Londres: ISTE éditions, 2018.

[8] "Notes on the Eclipse Plug-in Architecture", Eclipse.org, 2019. [Online]. Available: https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html. [Accessed: 28-Nov- 2019]

[9] R. Gronback, "Eclipse Modeling Project | The Eclipse Foundation", *Eclipse.org*, 2019. [Online]. Available: https://www.eclipse.org/modeling/emf/. [Accessed: 28- Nov- 2019]

[10] "Model Based Systems Engineering | Capella MBSE Tool", *Polarsys.org*, 2019. [Online]. Available: https://www.polarsys.org/capella/. [Accessed: 28- Nov- 2019]

[11] i. AG, "What are YAKINDU Statechart Tools?", Itemis.com, 2019. [Online]. Available: https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_yakindu_statechart_tools. [Accessed: 28- Nov- 2019]

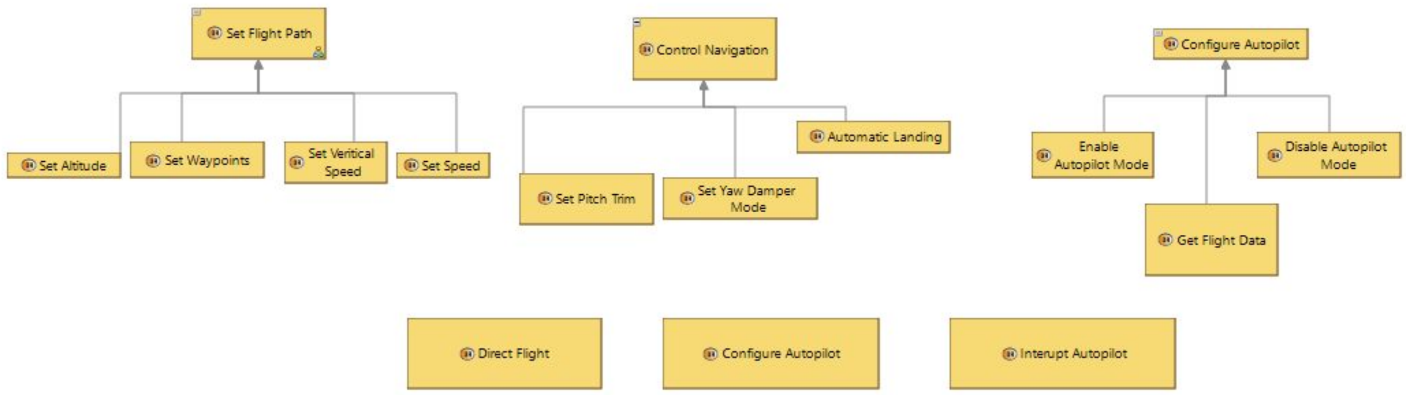
[12] "viatra/VIATRA-Generator", GitHub, 2019. [Online]. Available: https://github.com/viatra/VIATRA-Generator/wiki. [Accessed: 28- Nov- 2019]

Appendix

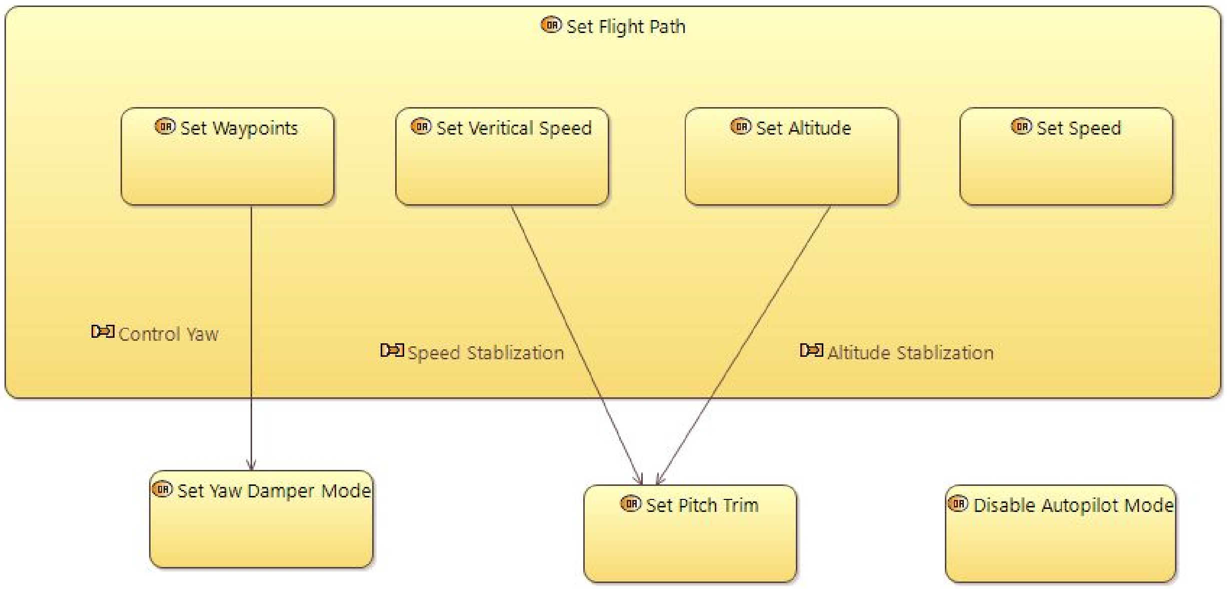
1. Capella Models

1.1 Operational Analysis

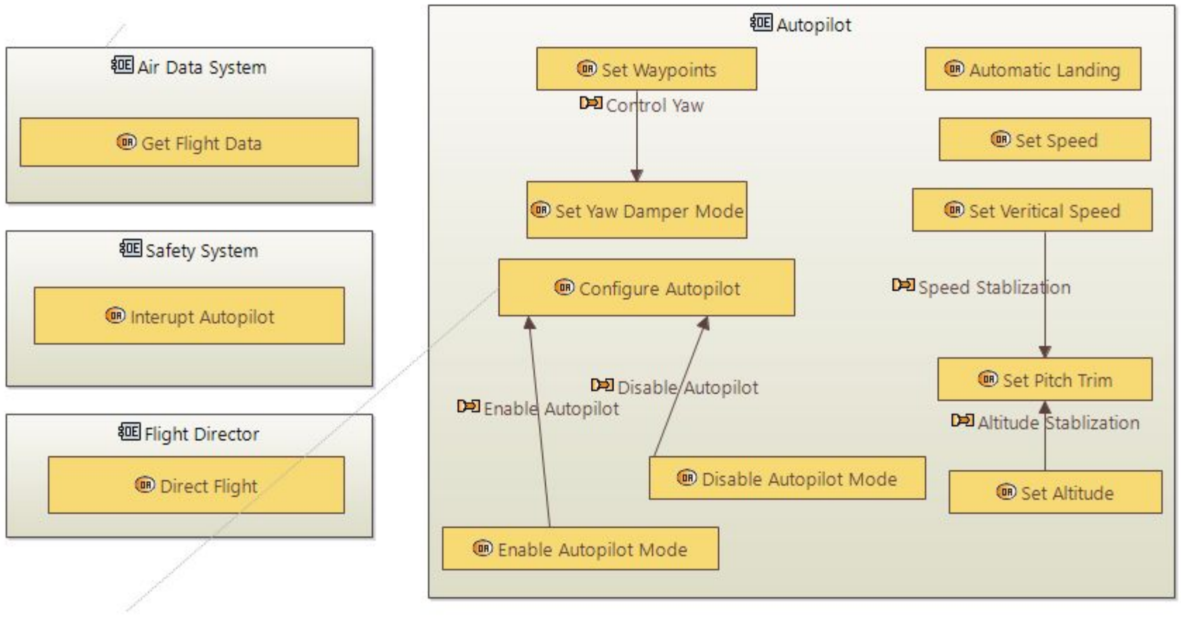
1.1.1 Operational Activity Breakdown Diagram (OABD)



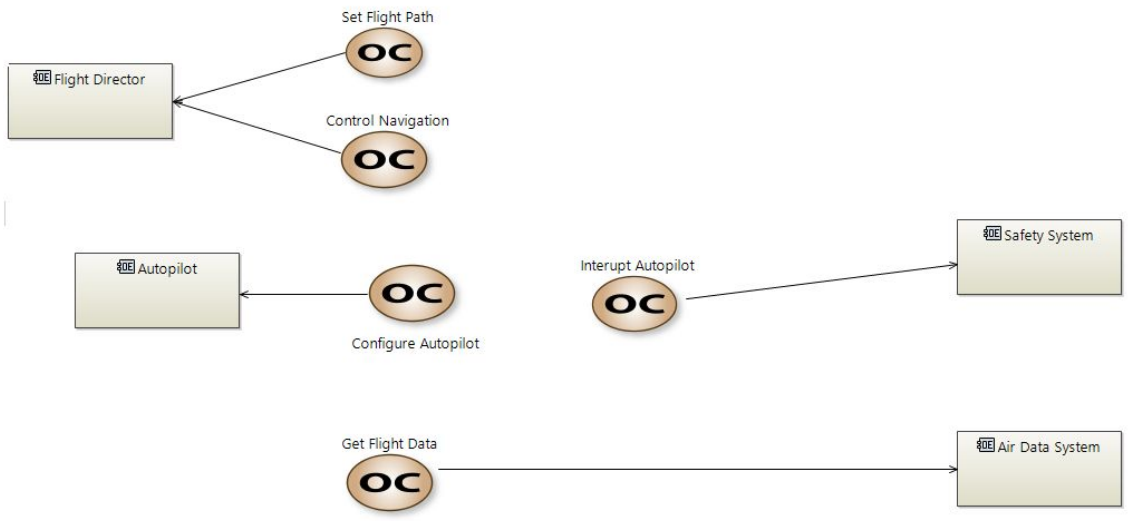
1.1.2 Operational Activity Interaction Blank (OAIB)



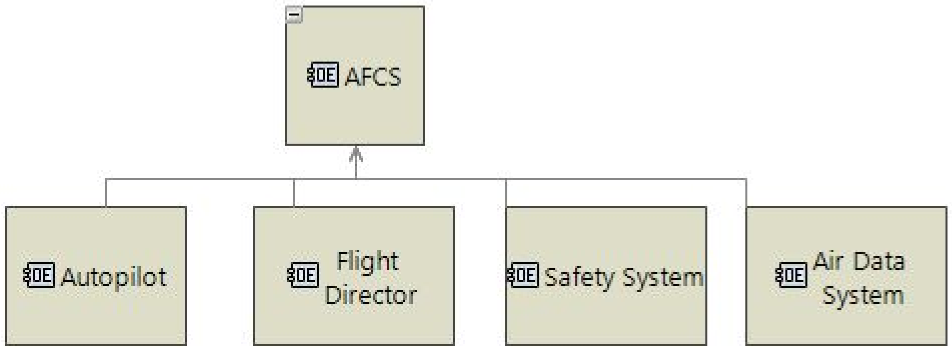
1.1.3 Operational Architecture Blank (OAB)



1.1.4 Operational Capabilities Blank (OCB)

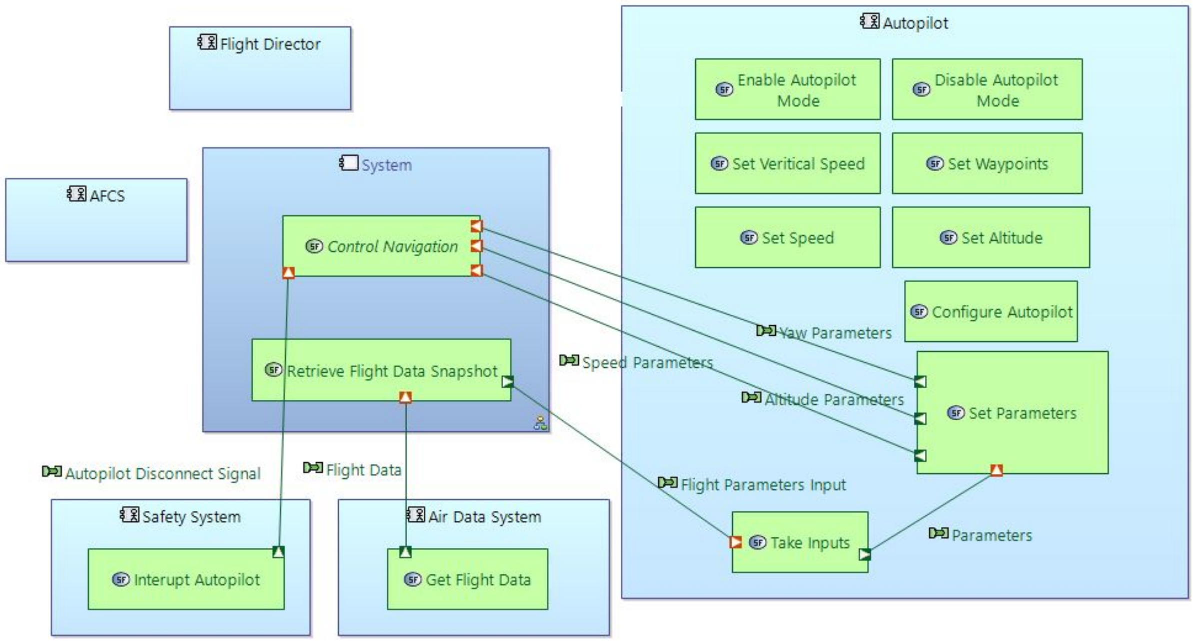


1.1.5 Operational Entity Breakdown (OEBD)

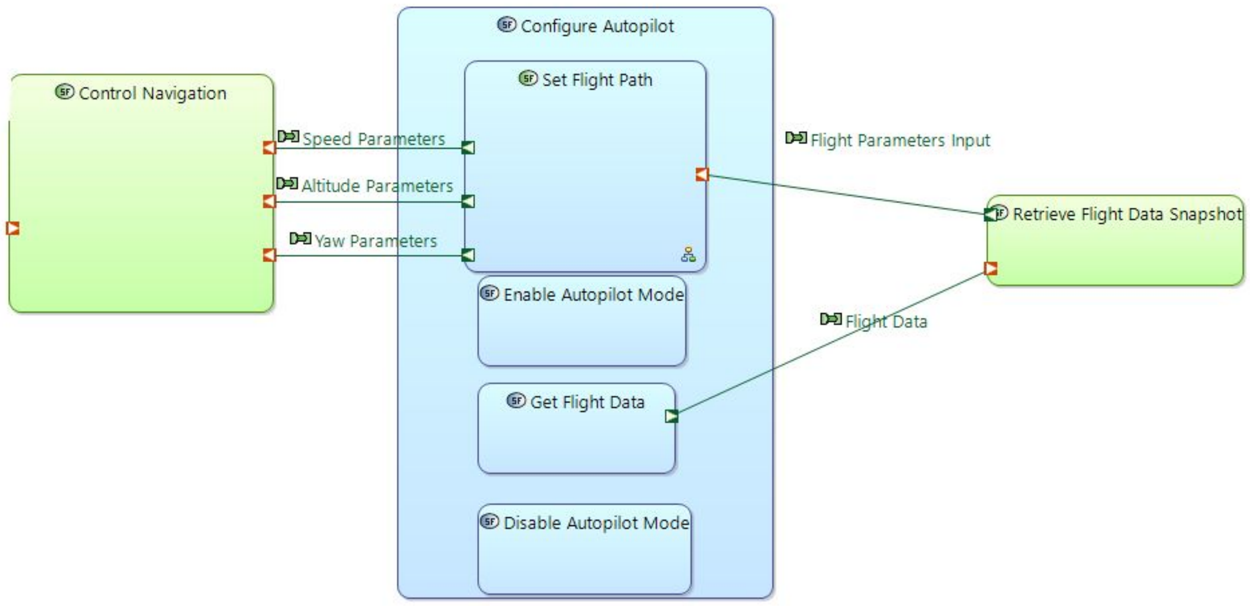


1.2 System Analysis

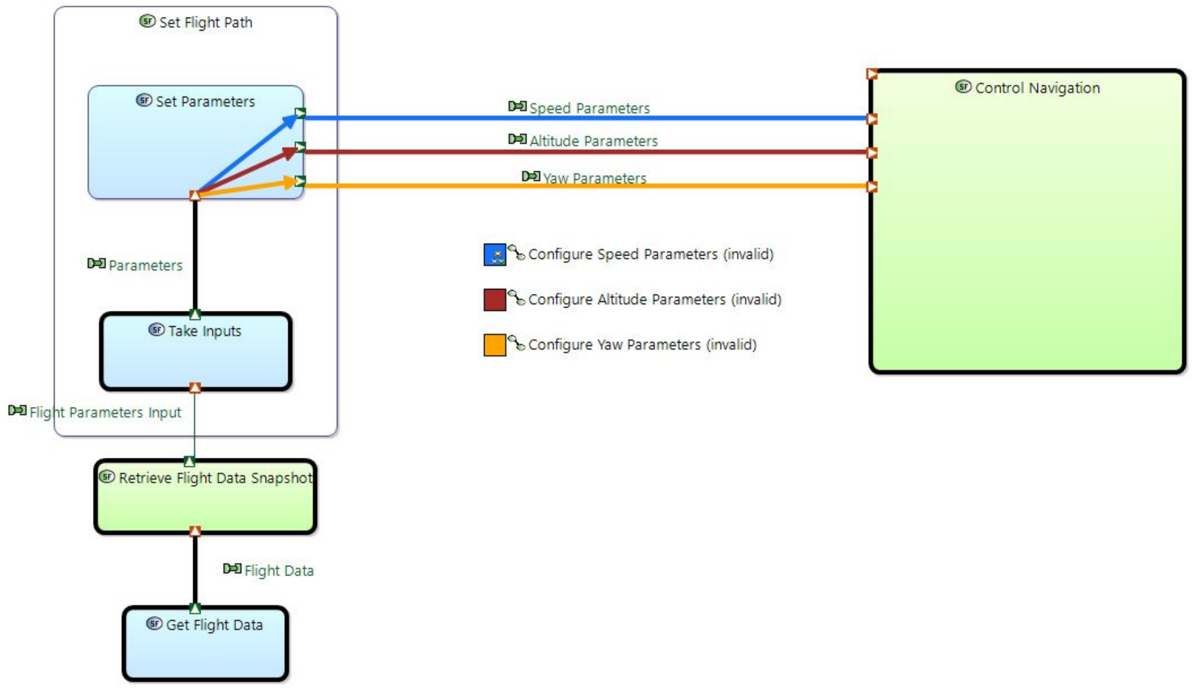
1.2.1 System Architecture Blank (SAB)



1.2.2.A System Data Flow Blank (SDFB)

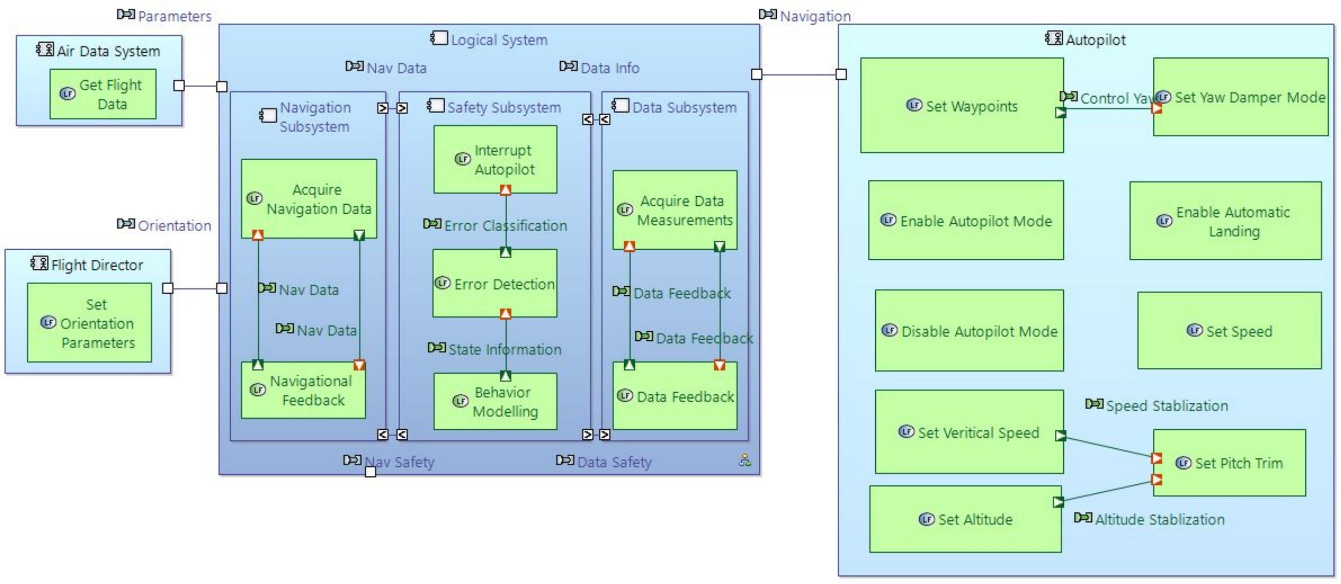


1.2.2.B System Data Flow Blank (SDFB)



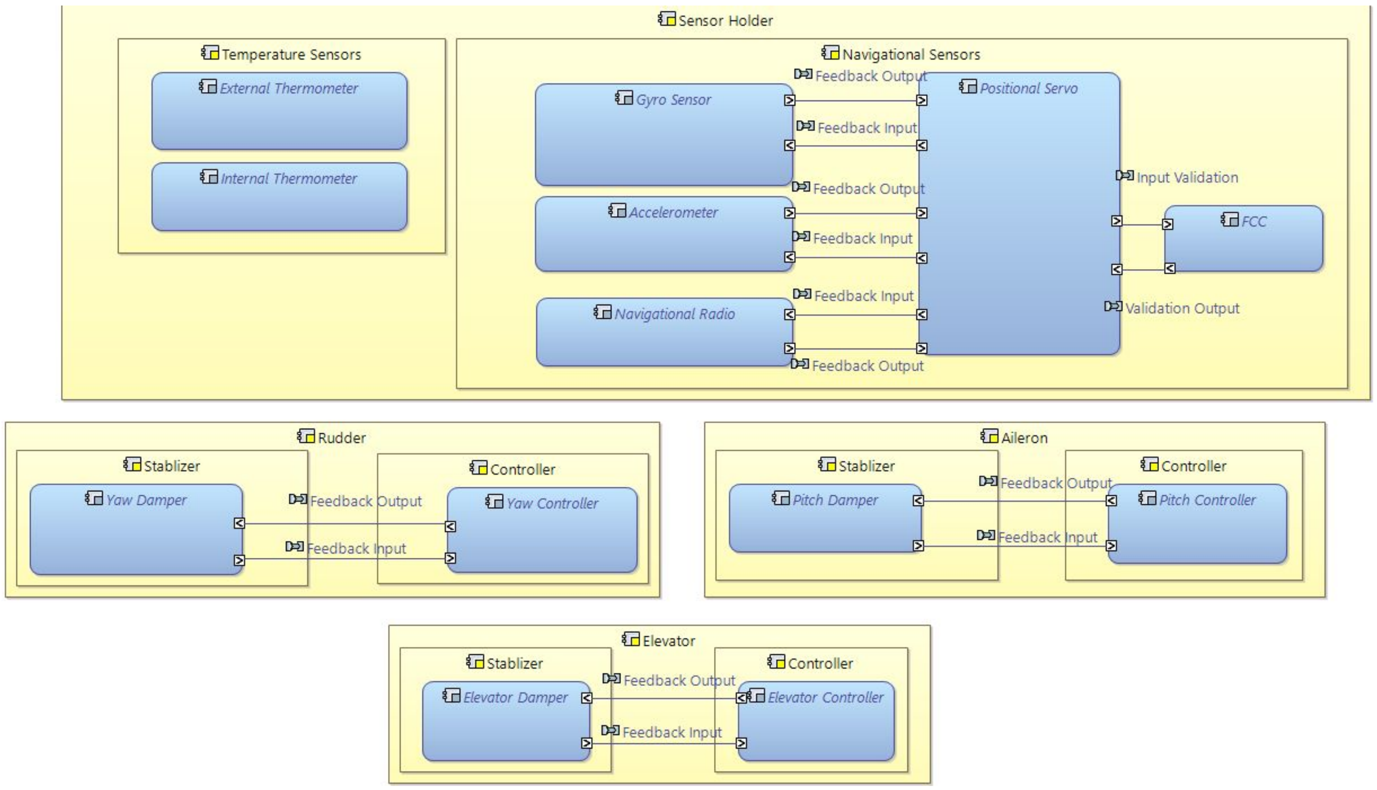
1.3 Logic Architecture

1.3.1 Logical Architecture Blank (LAB)



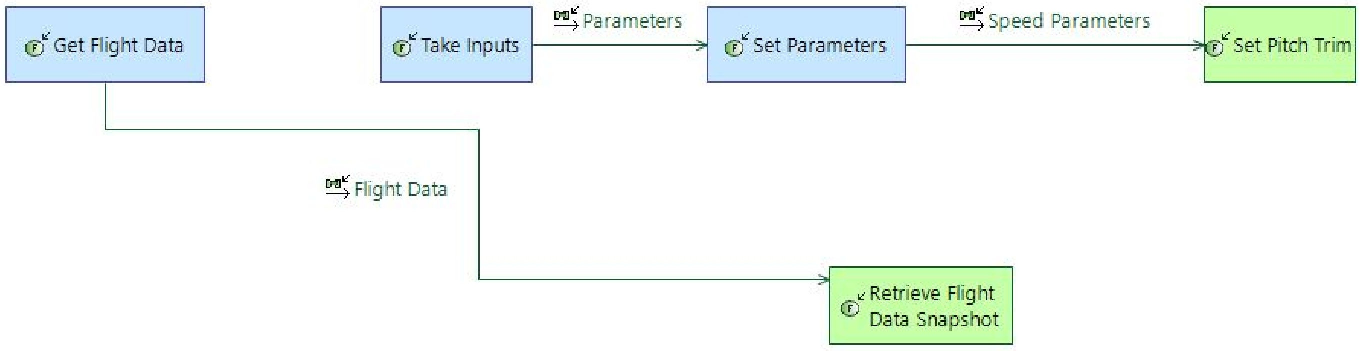
1.4 Physical Architecture

1.4.1 Physical Architecture Blank (PAB)



1.5 Common

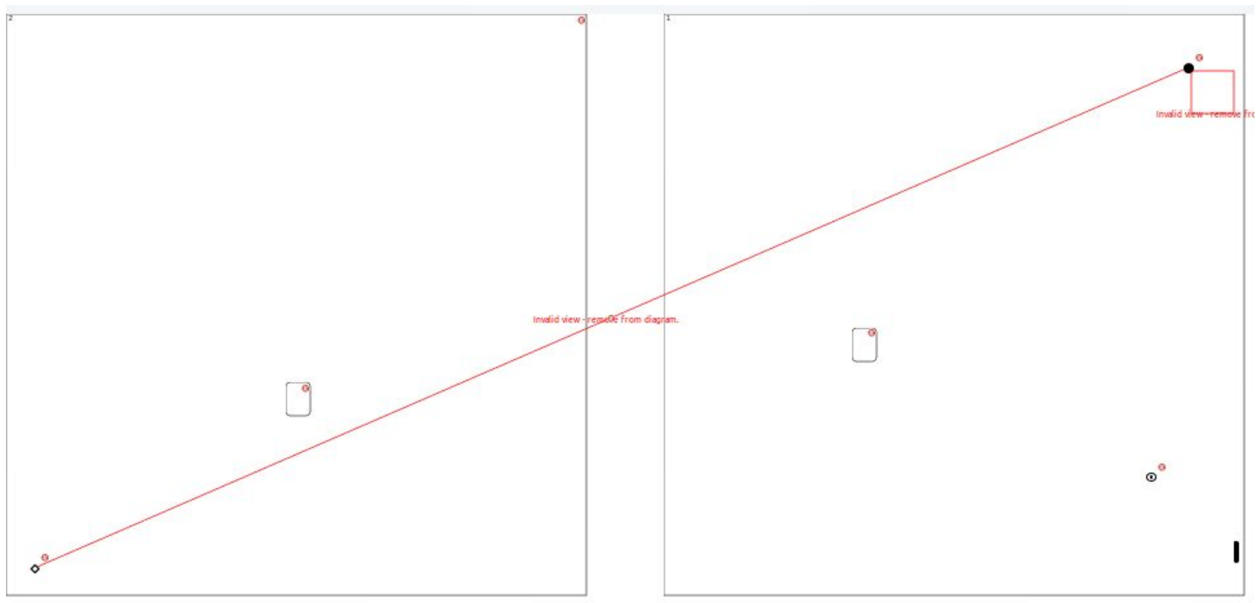
1.5.1 Functional Chain Description (SFCD)



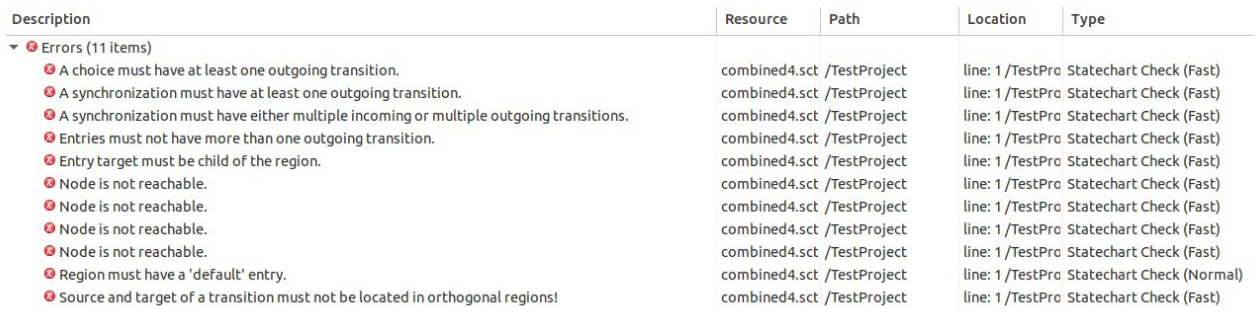
2. Generated Instance Models

2.1. Instance Model Examples

2.1.1. Example Generated Output Visualized


2
1
Invalid view - remove from diagram.
Invalid view - remove fro

2.1.2 Errors Reported for Generated Output



Description	Resource	Path	Location	Type
Errors (11 items)				
A choice must have at least one outgoing transition.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)
A synchronization must have at least one outgoing transition.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)
A synchronization must have either multiple incoming or multiple outgoing transitions.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)
Entries must not have more than one outgoing transition.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)
Entry target must be child of the region.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)
Node is not reachable.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)
Node is not reachable.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)
Node is not reachable.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)
Node is not reachable.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)
Region must have a 'default' entry.	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Normal)
Source and target of a transition must not be located in orthogonal regions!	combined4.sct	/TestProject	line: 1 /TestPro	Statechart Check (Fast)

2.1.3 Example Generator Output Instance Model

```
<?xml version="1.0" encoding="ASCII"?>
<yakindumm:Statechart xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:yakindumm="hu.bme.mit.inf.dslreasoner.domains.yakindu.sgraph">
  <regions>
        <vertices xsi:type="yakindumm:State"/>
        <vertices xsi:type="yakindumm:Synchronization"/>
        <vertices xsi:type="yakindumm:Entry"
incomingTransitions="//@regions.0/@vertices.2/@outgoingTransitions.1">
        <outgoingTransitions target="//@regions.1/@vertices.0"/>
        <outgoingTransitions target="//@regions.0/@vertices.2"/>
```

		</vertices>
		<vertices xsi:type="yakindumm:FinalState" choice="//@regions.1/@vertices.0"/>
  </regions>
  <regions>
		<vertices xsi:type="yakindumm:Choice"
incomingTransitions="//@regions.0/@vertices.2/@outgoingTransitions.0" finalstate="//@regions.0/@vertices.3"/>
		<vertices xsi:type="yakindumm:State"/>
  </regions>
</yakindumm:Statechart>

2.1.4 Example Post-Processed Instance Model

<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:notation="http://www.eclipse.org/gmf/runtime/1.0.2/notation"
xmlns:sgraph="http://www.yakindu.org/sct/sgraph/2.0.0">
  <sgraph:Statechart xmi:id="_XcFpUIAQEeq5St2JCc7KSQ" name="mauve">
		<regions xmi:id="_XcFpUYAQEeq5St2JCc7KSQ" name="complete">
		<vertices xsi:type="sgraph:State" xmi:id="_XcFpUoAQEeq5St2JCc7KSQ" name="purple"/>
		<vertices xsi:type="sgraph:Synchronization" xmi:id="_XcFpU4AQEeq5St2JCc7KSQ"/>
		<vertices xsi:type="sgraph:Entry" xmi:id="_XcFpVIAQEeq5St2JCc7KSQ"
incomingTransitions="_XcFpVoAQEeq5St2JCc7KSQ">
		<outgoingTransitions xmi:id="_XcFpVYAQEeq5St2JCc7KSQ"
target="_XcFpWYAQEeq5St2JCc7KSQ"/>
		<outgoingTransitions xmi:id="_XcFpVoAQEeq5St2JCc7KSQ" target="_XcFpVIAQEeq5St2JCc7KSQ"/>
		</vertices>
		<vertices xsi:type="sgraph:FinalState" xmi:id="_XcFpV4AQEeq5St2JCc7KSQ"/>
		</regions>
		<regions xmi:id="_XcFpWIAQEeq5St2JCc7KSQ" name="cutting">
		<vertices xsi:type="sgraph:Choice" xmi:id="_XcFpWYAQEeq5St2JCc7KSQ"
incomingTransitions="_XcFpVYAQEeq5St2JCc7KSQ"/>
		<vertices xsi:type="sgraph:State" xmi:id="_XcFpWoAQEeq5St2JCc7KSQ" name="blue"/>
		</regions>
  </sgraph:Statechart>
  <notation:Diagram xmi:id="_XcFpW4AQEeq5St2JCc7KSQ"
type="org.yakindu.sct.ui.editor.editor.StatechartDiagramEditor" element="_XcFpUIAQEeq5St2JCc7KSQ"
measurementUnit="Pixel">
		<children xmi:id="_XcFpXIAQEeq5St2JCc7KSQ" type="Region"
element="_XcFpWIAQEeq5St2JCc7KSQ">
		<children xsi:type="notation:Shape" xmi:id="_XcFpXYAQEeq5St2JCc7KSQ"
type="RegionCompartment">
		<children xmi:id="_XcFpXoAQEeq5St2JCc7KSQ" type="Choice"
element="_XcFpWYAQEeq5St2JCc7KSQ">
		<layoutConstraint xsi:type="notation:Bounds" xmi:id="_XcFpX4AQEeq5St2JCc7KSQ" x="25"
y="819"/>
		</children>
		<children xmi:id="_XcFpYIAQEeq5St2JCc7KSQ" type="State"
element="_XcFpWoAQEeq5St2JCc7KSQ">
		<children xsi:type="notation:Compartment" xmi:id="_XcFpYYAQEeq5St2JCc7KSQ"
type="StateTextCompartment"/>
		<layoutConstraint xsi:type="notation:Bounds" xmi:id="_XcFpYoAQEeq5St2JCc7KSQ" x="422"
y="539"/>

```
        </children>
        </children>
        <layoutConstraint xsi:type="notation:Bounds" xmi:id="_XcFpY4AQEeq5St2JCc7KSQ" x="1" y="1"
width="900" height="900"/>
        </children>
        <children xmi:id="_XcFpZIAQEeq5St2JCc7KSQ" type="Region"
element="_XcFpUYAQEeq5St2JCc7KSQ">
        <children xsi:type="notation:Shape" xmi:id="_XcFpZYAQEeq5St2JCc7KSQ"
type="RegionCompartment">
        <children xmi:id="_XcFpZoAQEeq5St2JCc7KSQ" type="State"
element="_XcFpUoAQEeq5St2JCc7KSQ">
        <children xsi:type="notation:Compartment" xmi:id="_XcFpZ4AQEeq5St2JCc7KSQ"
type="StateTextCompartment"/>
        <layoutConstraint xsi:type="notation:Bounds" xmi:id="_XcFpaIAQEeq5St2JCc7KSQ" x="280"
y="454"/>
        </children>
        <children xmi:id="_XcFpaYAQEeq5St2JCc7KSQ" type="Synchronization"
element="_XcFpU4AQEeq5St2JCc7KSQ">
        <layoutConstraint xsi:type="notation:Bounds" xmi:id="_XcFpaoAQEeq5St2JCc7KSQ" x="872"
y="785"/>
        </children>
        <children xmi:id="_XcFpa4AQEeq5St2JCc7KSQ" type="Entry"
element="_XcFpVIAQEeq5St2JCc7KSQ">
        <layoutConstraint xsi:type="notation:Bounds" xmi:id="_XcFpbIAQEeq5St2JCc7KSQ" x="795" y="45"/>
        </children>
        <children xmi:id="_XcFpbYAQEeq5St2JCc7KSQ" type="FinalState"
element="_XcFpV4AQEeq5St2JCc7KSQ">
        <layoutConstraint xsi:type="notation:Bounds" xmi:id="_XcFpboAQEeq5St2JCc7KSQ" x="737"
y="678"/>
        </children>
        </children>
        <layoutConstraint xsi:type="notation:Bounds" xmi:id="_XcFpb4AQEeq5St2JCc7KSQ" x="1" y="1"
width="900" height="900"/>
        </children>
        <edges xmi:id="_XcFpcIAQEeq5St2JCc7KSQ" element="_XcFpVoAQEeq5St2JCc7KSQ"
source="_XcFpa4AQEeq5St2JCc7KSQ" target="_XcFpa4AQEeq5St2JCc7KSQ">
        <bendpoints xsi:type="notation:RelativeBendpoints" xmi:id="_XcFpcYAQEeq5St2JCc7KSQ" points="[0,
0, 0, 0]$[0, 0, 0, 0]"/>
        </edges>
        <edges xmi:id="_XcFpcoAQEeq5St2JCc7KSQ" element="_XcFpVYAQEeq5St2JCc7KSQ"
source="_XcFpa4AQEeq5St2JCc7KSQ" target="_XcFpXoAQEeq5St2JCc7KSQ">
        <bendpoints xsi:type="notation:RelativeBendpoints" xmi:id="_XcFpc4AQEeq5St2JCc7KSQ" points="[0,
0, 0, 0]$[0, 0, 0, 0]"/>
        </edges>
  </notation:Diagram>
</xmi:XMI>
```