Modeling and Implementing Variability in Aerospace Systems Product Lines

by

Jesús Alejandro Padilla Gaeta

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Math
in
Computer Science

Waterloo, Ontario, Canada, 2014

Abstract

Avionics systems are becoming indispensable on both military and civil aircraft. As more and more of their vital functions are controlled by electronic devices, their system qualities such as safety, reliability, and fuel-efficiency are becoming increasingly dependent on the correct operation of the avionics on-board. As avionics applications increase in size and complexity, so are their development and testing costs. Therefore, it is not surprising that companies in this domain are looking for new development approaches that can help them deal with these challenges. In this regard, software product lines have been particularly successful. They can help reduce costs by designing systems that share a set of common assets that enable reuse in a structured way; they are developed as a family.

One asset that is particularly useful to handle complexity and that makes sense to approach as a family is models. They provide the means to communicate and get consensus with other stakeholders, scope the system to create, predict important properties or characteristics, among others. In other words, a good model can become the cornerstone of a successful system. However, if modeling one system is challenging, modeling a family of them becomes even harder. Each member can be different, so the model must be able to specify accurately what parts are common to all of them, what can vary, and under which conditions.

The purpose of this work is twofold. First, it evaluates the capabilities of SysML to accurately describe variability in the domain of avionics systems. Second, it provides guidelines for how to model and design systems that present it.

In the end, this work concluded that simple extensions allow SysML to be appropriate for describing systems with variability. Also, it introduced a methodology and a series of structural and behavioral patterns to describe families of systems while keeping their differences under control. Implementation patterns were also included to show how models can be connected with code. Finally, this whole approach was evaluated via a case study based on five real aircraft engine instances.

Acknowledgements

First, I want to thank my supervisor, Dr. Krzysztof Czarnecki, for giving me this incredible opportunity. Thank you for sharing your knowledge, for your patience and guidance, and especially for your trust. It has been a remarkable experience that I will always remember.

Next, I would like to acknowledge Mike Darby and everybody at Pratt. Thank you for your patience, your time, and your support. This internship was fantastic; I learned so much from all of you.

Thank you to my dear friends at the GSD Lab, Alexandr Murashkin, Ed Zulkoski, Jianmei Guo, Leonardo Passos, Michal Antkiewicz, Pavel Valov, Rafael Olaechea, Wenbin Ji, and Zubair Akhtar. We do have the best lab at the university.

I would like to express my deepest gratitude to my parents and my sister. I really appreciate all your love and support. Everything I've accomplished is because of you. Thank you very much.

Finally, I would like to thank all the wonderful people that I have met in Canada, who have made the time I spent in this program an experience impossible to forget.

Dedication

This is dedicated to my loving parents and sister, who have been my guidance, my solace, my source of strength, and my biggest fans.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Safety-critical embedded systems [41] are becoming more and more pervasive in the modern society. They are present in a vast amount of domains, which include the automotive, avionics, and medical. One of the characteristics that many of these systems share is that any misoperation can produce serious consequences, like enormous economic loss, or even putting human lives at risk. Therefore, companies must ensure that their products are safe and reliable. In the avionics domain in particular, they also have to be certified by national authorities; this process can be costly and time-consuming.

As avionics applications increase in size and complexity, so are their development and testing costs. Therefore, it is not surprising that companies in this domain are looking for new development approaches that can help them deal with these challenges. In this regard, software product lines have been particularly attractive [35]. They can help reduce costs by designing systems that share a set of common assets that enable reuse in a systematic way; they are developed as a family.

One type of assets that is particularly useful to manage complexity is models. They provide the means to communicate and get consensus with other stakeholders, scope the system to create, predict important properties or characteristics, among others. In other words, a good model can become the cornerstone of a successful system. However, if modeling one system is challenging, modeling a family of them becomes even harder. Each member can be different, so models must be able to specify accurately what parts are common to all of them, what can vary, and under which conditions.

In order to obtain the benefits of modeling, models must be expressed in a notation that is well-understood by all stakeholders. Establishing such consensus can be quite challenging, and thus reliance on standards can be beneficial. A standard that is particularly relevant in the avionics domain is SysML [32]. This is because it is based on UML, which is already well established; many existing tools already support it; and it can be extended as necessary.

The purpose of this work is twofold. First, it concentrates on evaluating the capabilities of SysML for describing accurately models with variability. Second, it attempts to create patterns and a methodology to serve as a guideline for the modeling and development of avionic product lines, that is, product lines of aviation electronic systems.

The contributions of this work are as follows. First, it assesses the ability of SysML to describe systems with variability. Second, it defines modeling patterns to enable the description of variability. Next, it describes implementation patterns necessary to connect models with code, expressed as SCADE models, which are common in the avionics domain. Finally, it provides the evaluation of this approach with a model of a real family of systems.

1.1 Methodology

The development and evaluation of this work were performed via two case studies. The first consisted of the propeller example discussed in Chapter 3. Although it contains realistic data, this study is not based on a real system, but on domain knowledge from literature for the purpose of an illustrative example. Nonetheless, it allowed identifying the types of variabilities that occur in avionic systems, how variability affects the modeling task, and it provided enough data to formulate the methodology and the patterns described in the following sections.

The second case study, on the other hand, was based on real data provided by our industrial partner, Pratt & Whitney Canada [36]. Based on this information, a family architecture model of a fuel control unit was created, and evaluated by accommodating five real engine instances. This approach allowed the validation of the material presented in this work. Chapter 8 presents the results.

1.2 Thesis Organization

Chapter 2 presents a basic introduction to SysML, the diagrams it provides and compares it to other modeling standards. Although it does not describe the notation and semantics of the language, some of that information is contained in Appendix A.

Section 3 provides background information about turbo engines and introduces the running example that is used throughout this work.

Chapter 4 presents the methodology to model a software product line in the avionics domain. Some of the steps described there require using patterns, which are presented in sections 5, 6 and 7.

The evaluation results are shown in Chapter 8, while sections 9 and 10 present the related work and the conclusions. Finally, Appendix B contains the suggested SysML variability profile.

Chapter 2

SysML and Related Notations

2.1 Introduction

SysML is a general-purpose graphical modeling language for system engineering applications supported and maintained by the Object Management Group (OMG) [33, 32]. It allows the specification, analysis, design, verification, and validation of a broad range of complex systems, which include hardware, software, information, processes, personnel, and facilities.

Instead of focusing on documents, SysML supports the so called Model-Based System Engineering (MBSE) [25, 15] approach that concentrates on defining a consistent and structured set of views, stored and managed in a repository; they make up the model of the system. This organization allows designers to manage complexity since each diagram provides an abstracted view of the system or part of it. Note that SysML is just a visual modeling language, not a methodology. In fact, to remain widely applicable, it is completely vendor and methodology agnostic.

SysML language was designed as an extension (profile) of UML 2. It reuses a subset of the constructs existing in the UML metamodel, modifies the parts that are too software oriented, and provides extensions necessary to meet the system engineering needs. However, it remains compatible with UML, so existing modeling tools can easily support it.

Figure 2.1 shows the relationship between UML and SysML as a Venn diagram. The region of the SysML set that does not intersect represents the extensions included in the language. On the other hand, the section where both sets meet signifies the parts that are

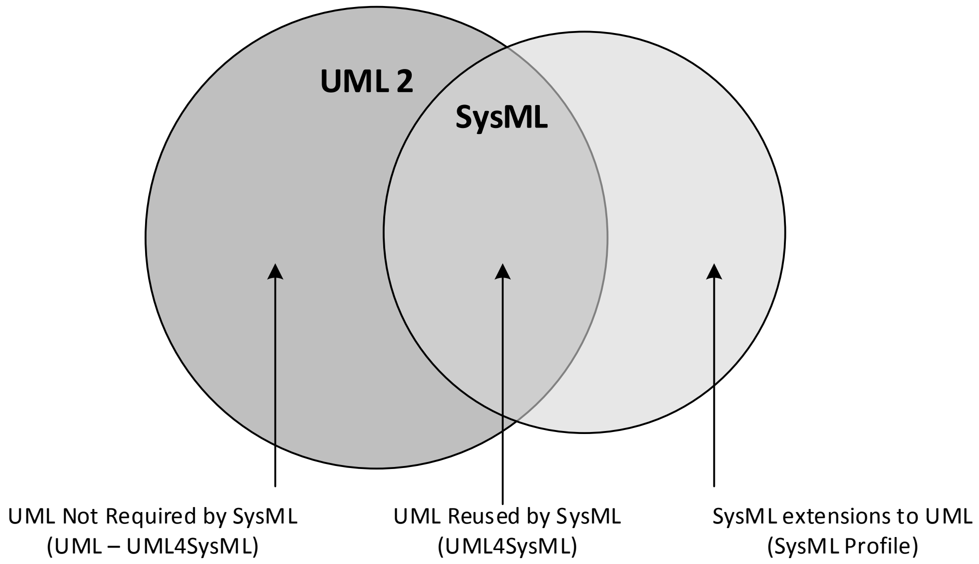


Figure 2.1: Relationship between UML 2 and SysML.

common to both languages, and that are contained in the so-called *UML4SysML* subset. The rest contains the components of UML 2 not necessary for systems engineering that are removed from SysML altogether.

2.2 Diagrams

This section provides a brief overview of the diagrams present in SysML, shown in figure 2.2. Check Appendix A or SysML specification [32] for more information about their notation.

- **Activity Diagram:** This type of diagram allows describing behavior in terms of flows of inputs, outputs, and control. They specify a sequence of steps needed to transform the provided inputs into expected outputs. In essence, they are very expressive flow charts.

- **Sequence Diagram:** This type of diagram presents behavior as a sequence of messages exchanged between elements. Such interactions can happen internally, between parts of the system, or externally, between the system and its environment.

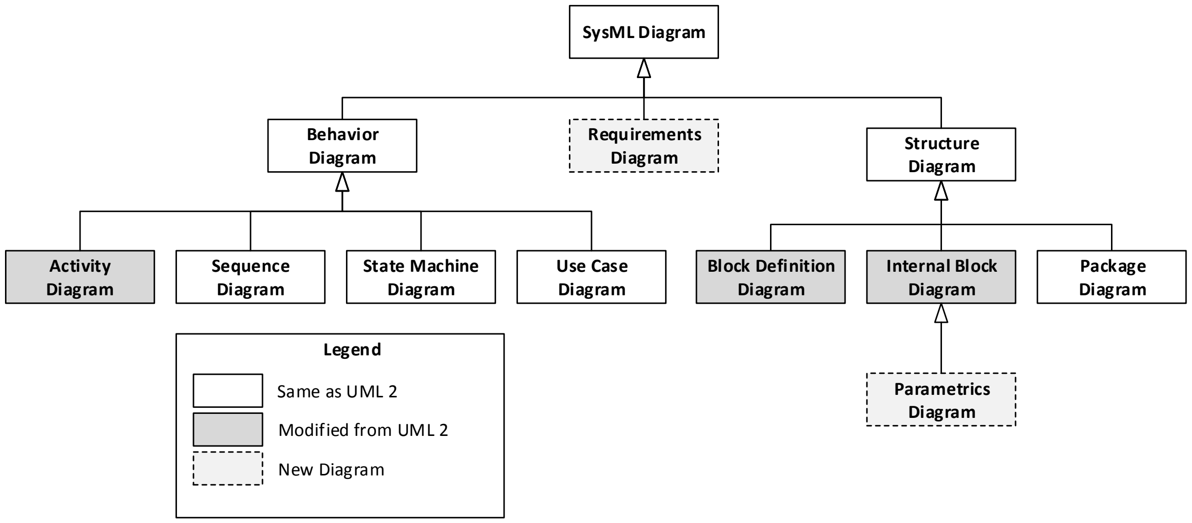


Figure 2.2: SysML diagrams.

- **State Machine Diagram:** This type of diagrams define the state-based behavior of a block throughout its life cycle in terms of system states and their transitions.
- **Use Case Diagram:** This type of diagrams describe the high-level functionality and uses of a system, which will be further described by the other behavioral diagrams.
- **Block Definition Diagram:** Blocks in SysML are modular units of system description. Each one of them describes a system, a part, or another element of interest. The Block Definition Diagram (BDD) specifies the structure of the system as a group of blocks, and their relationships such as association, generalization, and dependencies.
- **Internal Block Diagram:** This type of diagrams, also referred to as IBD, describe the internal structure of a block in terms of properties, connectors, and ports. They are also useful to model flows in the system.
- **Package Diagram:** This type of diagrams display the organization of the model in terms of packages that contain other elements, and their dependencies.
- **Parametrics Diagram:** This type of diagram describes constraints between the block properties. It provides the means to perform engineering analysis models such

as performance and reliability, among others.

- **Requirement Diagram:** This type of diagram presents text-based requirements organized in a hierarchical structure suited for traceability.

2.3 SysML and AADL

The Architecture Analysis & Design Language (AADL) [19, 46] is a standard created by the Society of Automotive Engineers (SAE) [38] that provides formal modeling concepts for the specification, analysis, and automated integration of computer systems. It supports model-based development approach and is particularly useful for complex real-time embedded systems.

Since SysML and AADL provide similar features and both can be used to model avionics applications; each one has different pros and cons. For example, AADL has been designed from scratch to support performance-critical systems, so it contains constructs and concepts necessary to model hardware and software, and includes most of the information needed in avionics applications. SysML, on the other hand, is a general-purpose modeling language, and as such it lacks some of these constructs out of the box. In particular, it does not contain a comprehensive model of time, and thus is unable to portray timing related data without extending it. However, SysML is based in UML, so it supports other profiles, like MARTE, which can be added to address these limitations. Furthermore, existing UML tools can be easily extended to support SysML; this is probably the biggest benefit over AADL.

Since AADL contains useful concepts, there have been attempts to integrate it with SysML. One of these experiments tried to adopt some of its ideas via profiles [7]. Although interesting, this is beyond the scope of this work.

2.4 SysML and MARTE

Just like SysML, MARTE [30, 39] is an extension of UML. It focuses on adding capabilities to model real-time and embedded software of cyber-physical systems. It adds the ability to define and specify in a precise manner quantitative and qualitative measures required for advanced analysis. Also, it provides the ability to describe resources specific to real-time that span from hardware, like memory, processors, and networks, to software elements, such as threads and mutexes. Finally, it also introduces a comprehensive model of time.

Since MARTE is also a UML profile, it can be incorporated to SysML to address some of its limitations. However, this must be done with caution; they have some overlapping constructs that can introduce ambiguity or some other problems, such as conflicts. Fortunately, MARTE has been structured in a modular way, so particular pieces of the profile can be chosen [14].

2.5 SysML and SCADE

SCADE Suite [42] is a model-based development environment, developed by Esterel Technologies [16], which allows developing software for safety-critical systems. One of the biggest benefits that it provides to the avionics domain is that it contains a code generator called KCG [1] that produces C code tailored for embedded applications. Furthermore, KCG has been qualified according to different safety standards, such as DO-178B [12] and C [13]; this simplifies the certification process as it eliminates the need to perform reviews of the generated code.

One important aspect to highlight is that SCADE Suite introduces a new language that is adequate for developing safety-critical software. However, it is not designed to describe system architectures; modeling languages, like SysML, are better suited for this task. Esterel recognized this problem and introduced SCADE System [43]. This tool allows the modeling of systems using a small subset of SysML. However, since these languages are quite different, the connection is still incomplete. More effort is necessary to provide a consistent and comprehensive bridge between SysML and SCADE Suite. This work provides some insight about this connection from the variability point of view.

2.6 Summary

This chapter introduced the modeling language SysML. It explained that SysML is an extension to UML and described how they relate to each other. Also, it presented the diagrams that make up SysML. Finally, it provided a comparison between SysML and other modeling languages, such as AADL and MARTE, as well as other technologies like SCADE.

Chapter 3

Propeller Example

3.1 Background

Turbine engines represent one of the most popular types of powerplants used in modern airplanes. They produce thrust by increasing the velocity of air flowing through them. Although their structure can vary, they typically contain an air inlet, compressor, combustion chambers, a turbine section, and exhaust [18]. Depending on how they provide power, they are classified as follows:

- **Turbojet** engines compress air and ignite the air-fuel mixture to produce exhaust gases that will create thrust. Unfortunately, they are not very efficient, so they have been slowly replaced by the other types of engines.
- **Turbofan** engines are very similar to turbojets, they add a fan section in front of the compressors, which is driven by the turbine. Even though the blades of the fan rotate at much slower speed than the rest of the engine, they can move large masses of air producing an enormous amount of thrust.
- **Turboprop** engines couple the turbine to an aircraft propeller. The power produced by the engine is used to drive the propeller, which will transform the rotatory motion into thrust. This type of engines is adequate for low-altitude, subsonic flights.
- **Turboshaft** engines are typically used by helicopters. Just like turboprops, they provide power to a shaft that operates a rotor. However, most of the energy is used to drive a turbine rather than produce thrust.

The examples in this work touch upon some features and concepts of the last three types; turbojet engines are never mentioned.

Finally, it is important to note that although it is possible to have an entirely mechanical system to control a turbo engine, the examples described in this work contain an electronic engine controller (EEC) to manage and orchestrate the whole operation of the engine.

3.2 Description

The case study presented in this section models a constant-speed controller for a turboprop engine. This type of system ensures that the propeller maintains an optimum speed, to provide the necessary thrust and to be fuel efficient, and can be accomplished by modifying the pitch of the blades. These are the main features relevant in this example:

- **Reversing:** This is an optional mode that enables the engine to provide negative thrust; it can allow the aircraft to decelerate.
- **Open or Closed Loop:** The control of the pitch of the propeller blades can be performed in a closed loop if there is a sensor providing feedback about the current position. Otherwise, the controller must leverage other data in the engine, like current propeller speed, to manage this value.
- **Synchrophasing:** It allows the pilot to reduce the noise and vibration produced by two engines by matching their propeller speed and phase. This feature only makes sense in multi-engine installations.
- **Anti-Ice:** Ice formation can significantly disturb the operation of a propeller; in extreme cases it can even endanger the whole aircraft operation. Therefore, engines contain anti-icing systems to prevent this accumulation from happening. There are many types of them, like fluid-based, electric, or hot-air-based.

Figures 3.1 and 3.2 present the top-level decomposition of the engine in block definition (BDD) and internal block (IBD) diagrams.

3.3 Summary

This chapter provided a basic introduction to turbo engines, and described its main types; namely turbojet, turboprop, turbofan, and turboshaft engines. Also, it gave a basic description of the case study used throughout the rest of this work. In particular, it outlined the features that make up the variability in the example.

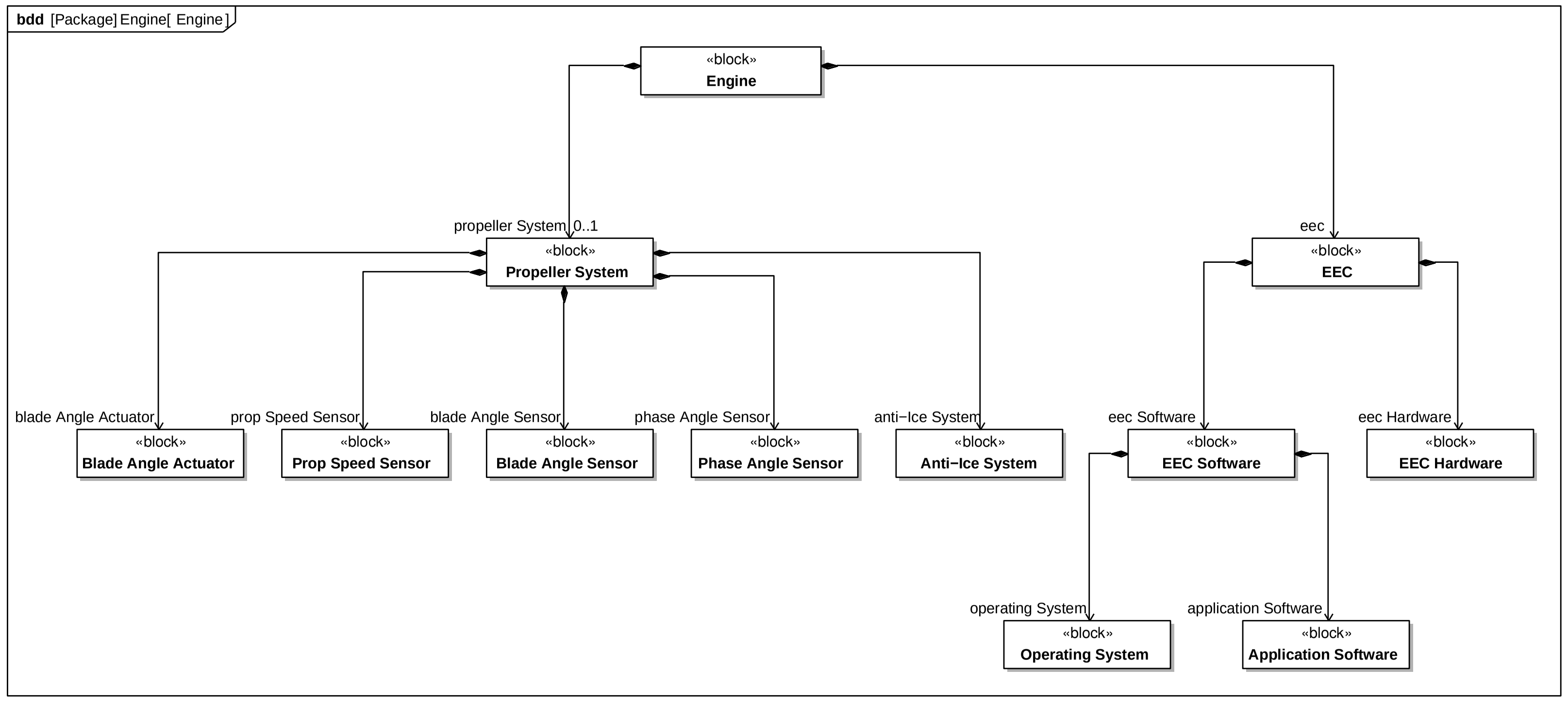


Figure 3.1: BDD of top level decomposition of engine.

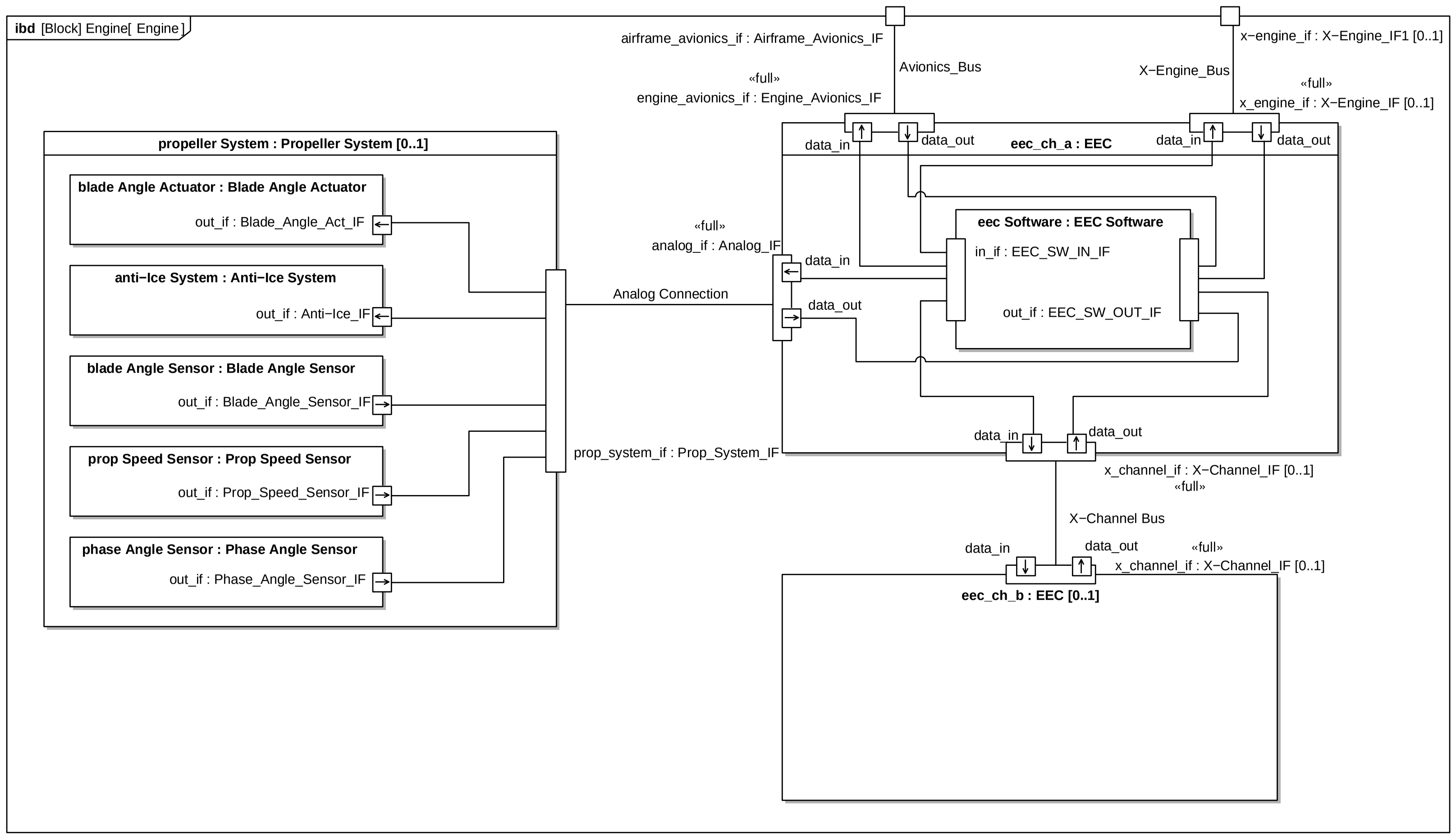

ibd [Block] Engine[Engine]
airframe_avionics_if : Airframe_Avionics_IF
Avionics_Bus
«full»
engine_avionics_if : Engine_Avionics_IF
X−Engine_Bus
x−engine_if : X−Engine_IF1 [0..1]
«full»
x_engine_if : X−Engine_IF [0..1]
data_in
data_out
eec_ch_a : EEC
data_in
data_out
propeller System : Propeller System [0..1]
blade Angle Actuator : Blade Angle Actuator
out_if : Blade_Angle_Act_IF
anti−Ice System : Anti−Ice System
out_if : Anti−Ice_IF
blade Angle Sensor : Blade Angle Sensor
out_if : Blade_Angle_Sensor_IF
prop Speed Sensor : Prop Speed Sensor
out_if : Prop_Speed_Sensor_IF
phase Angle Sensor : Phase Angle Sensor
out_if : Phase_Angle_Sensor_IF
«full»
analog_if : Analog_IF
Analog Connection
prop_system_if : Prop_System_IF
data_in
data_out
eec Software : EEC Software
in_if : EEC_SW_IN_IF
out_if : EEC_SW_OUT_IF
data_in
data_out
x_channel_if : X−Channel_IF [0..1]
«full»
X−Channel Bus
data_in
data_out
«full»
x_channel_if : X−Channel_IF [0..1]
eec_ch_b : EEC [0..1]

Figure 3.2: IBD of engine internal structure.

Chapter 4

Methodology for Modeling a Software Product Line in Avionics

Avionics systems are becoming indispensable on both military and civil aircrafts. As more and more of their vital functions are controlled by electronic devices, system qualities such as safety, reliability, and fuel-efficiency are becoming increasingly dependent on the correct operation of the avionics on-board [40]. However, at the same time, these systems are growing in size and complexity, which makes them even harder to develop. Hence, it is clear that a rigorous development process is required to cope with these challenges. A good architectural model can aid in this endeavor, especially if these systems are to be developed in a software product line. This chapter describes the methodology that we propose for creating such a model using SysML.

4.1 Goals

Before any software can be used in any airborne system, companies must ensure that it is functioning correctly and that it is compliant with airworthiness requirements. For this purpose, the software needs to be approved by national certification authorities; this process can be lengthy and time consuming. Therefore, companies often follow guidelines, like DO-178B and C [12, 13], for the development of their products. These standards do not focus only on the correct functioning of the final software but on a rigorous process being followed during the whole lifecycle. Pervasive traceability can be used to fulfill this requirement, which is why it is one of the primary goals of our proposed methodology.

It is important to remark that although these standards provide guidance of the overall process to follow, they do not specify any particular development methodology or technology; each organization is responsible for adjusting the process to their business needs and work procedures. Therefore, the methodology being proposed in this work should not be rigid; it must be flexible to allow being adapted to each of these different environments.

Another guiding factor is ease of use and maintainability. The architectural models are useful only if the information they contain is accurate and up to date, which is possible only if they are regularly maintained. If the effort required to keep them updated is too high, there is a risk that the models will be abandoned, or even worse, used with outdated information. For this reason, the methodology tries to streamline the work as much as possible, and also highlights the areas where tool support and automation could reduce time and effort.

Even though this work was envisioned for avionics it does not preclude it from being applied to other domains, but it would probably need to be adapted as required. In particular, some goals, like pervasive traceability, might need to be relaxed to make it more usable and practical for a specific domain.

4.2 Contributions

The methodology described in this section is not entirely new. It builds upon existing work in systems modeling [9, 29, 21], software product lines [8], and aerospace system development practices [3], and adapts them to modeling variability in avionics systems. Among its contributions, it introduces the concept of family and instance architectures and establishes a way of connecting the models via inheritance to keep them synchronized. Second, it proposes a way to enable requirements variability. Third, it explains how to provide pervasive traceability using existing SysML constructs. Finally, it describes how to keep the model consistent with the use of constraints.

4.3 Description

The core of the methodology relies on dividing the model into two parts, the family model and the instance models. The first, as the name implies, describes the architecture of all the products of the family; it highlights which parts are common and which parts can vary. In other words, this model contains variability. Analogously, the second describes the design of only one product of the family; it illustrates the structure of the instance obtained by configuring the original model and by creating new parts exclusive to the current instance. It is very likely that this model will contain very little or no variability.

4.3.1 Family Architectural Model

This section describes the steps that are required to create the family architectural model. It is important to highlight that although they are thought to be applied in sequence, the reality is that they are part of an iterative process; some steps might be executed out of order, and may be repeated as many times as required. What is important is to execute all of them, since their output is necessary for creating a complete, traceable, and consistent architecture.

1. **Create Use Cases**
   The first step is to identify the high-level functionality of the family of products to be modeled. Use cases are suitable for this task, since they describe the functionality of the system in terms of how it is used to achieve the goals of its various users [21]. They are adequate to explore the operation of particular instances of the family and new or unknown features of the software product line. However, because of their narrow scope, they are not good at capturing product-line-wide variability and commonalities [8]; this is fine because the objective of this stage is not to provide detailed information. On the contrary, the key idea is to identify the system, the actors, and the external systems, and describe how they interact with each other at a very high-level of abstraction. In other words, the focus must be on specifying what the intended scope of the system is.

2. **Define Feature Model**
   Part of the process of scoping the system is to find which functionality is common to all instances of the family, and which one is variable. One useful way to capture this information is in terms of features, which can be thought as user-visible aspects, quality, or characteristics of a system [26, 8]. For example, a company might offer aircraft engines with or without a propeller, and they can be reversing or non-reversing; each of these options is a feature. This information can then be organized and documented in feature models (FM), as shown in figure 4.1. As mentioned before, feature models in this document are presented in Clafer [5] language.

3. **Define System and Software-Level Requirements**
   At this point, the functionality of the system must be decomposed and formalized

```
abstract AircraftSystem
    Configuration : SystemConfiguration
    Airframe
        LocalEngine : Engine

abstract SystemConfiguration
    NumberOfEngines : integer
    [ NumberOfEngines > 0 && NumberOfEngines <= 2 ]
    Communication
        xor Avionics
            ARINC
        xor XEngine ?
            CANBUS
        [ NumberOfEngines > 1 <=> XEngine ]
        xor XChannel
            CANBUS
    Propeller
        Synchrophasing ?
        [ NumberOfEngines > 1 <=> Synchrophasing ]
        xor PropellerType
            Reversing
            NonReversing
        xor Feedback
            Open_Loop
            Closed_Loop
```
Figure 4.1: Part of the Feature Model (FM) of an aircraft system.

in requirements. As described in the SysML specification [32], each of these is a text-based description that specifies a capability or condition that must be satisfied, a function that the system must perform, or a performance condition to be achieved. One of the benefits of using SysML diagrams (or tables) is that their structure enables adding traceability and variability handling information, allowing requirements to be in-sync with the rest of the model.

The organization of requirements must follow a hierarchical structure. The ones located at the top provide information about the system as a whole. Each of them is allocated conceptually to hardware or software depending on which of the two will be in charge of satisfying them; this is consistent with the standard SAE ARP4754 [3]. At the software level, new requirements are created to describe each system-level requirement at a finer level of granularity; multiple levels of decomposition are allowed as long as clear traceability is always present. When considered detailed enough, requirements are allocated to software components. It is important to remark that this methodology makes special emphasis on software. Requirements allocated to hardware may be handled differently, but this lays outside the scope of this work.

The requirements specified in the family architecture model may encompass a wide variety of products. Since instances are expected to be different from each other, it is also expected that not all requirements will be the same for all variants. In fact, the most likely situation is that a set of them will be present in all products, some will be optional, while others will need to be customized by each instance. To provide guidance about how to use each requirement, they must be classified with one of the following stereotypes:

- **Mandatory Requirement:** All instances must include them, but cannot modify them.
- **Optional Requirement:** Instances cannot modify them, but can decide whether to include them or not.
- **Variant Specific Requirement:** Any instance must always include requirements with this stereotype, but it must customize them first.
- **Optional Variant Specific Requirement:** Instances can decide where to include them or not. However, they must be customized before being used.

Finally, the inclusion of a requirement might be controlled by the presence of a feature in the feature model. If that is the case, the requirement must be tagged with the corresponding feature ID[1].

4. **Modeling Structure and Behavior**
   This step deals with modeling of the family architecture. This endeavor consists of two parts, the structural and the behavioral. The first one handles static aspects, like the components (blocks) that make up the system, how they connect with each other, and which data flows between them. The latter deals with the dynamic aspects of the system: how information is processed to produce usable output, how messages are sent between blocks to respond to some stimuli, among others. Both of them are important, as they show different facets of the system.

   Although there are no strict rules to guide whether to model structure or behavior first, experience suggests that it might be a good idea to start with the behavioral part. This approach can help understand requirements better, can aid in solving ambiguity, and will assist in the selection of a particular structural decomposition to use later on. However, regardless of the order chosen, both parts must be connected to each other via allocation.

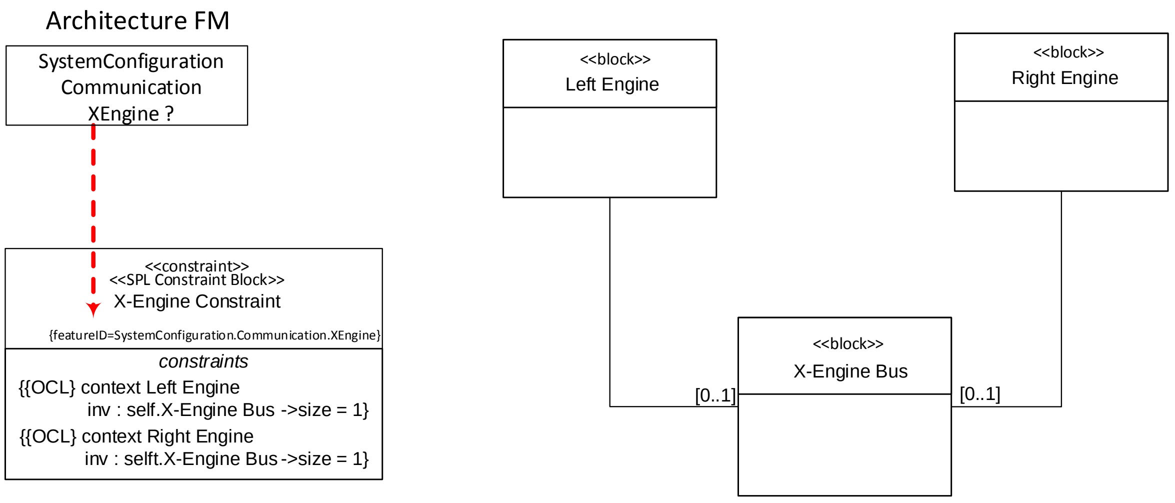
   Models created during this stage must depict precisely how the feature variations can impact the structure and behavior of the system. This work provides a series of patterns and practices to aid in this process; following chapters describe them in further detail.

5. **Constraints Definition**
   As mentioned before, the structure and behavior of the system can vary depending on the feature selection. However, regardless of the actual choices, instance models must always be consistent. As the size of the system increases, this process can become quite challenging. Therefore, there must be a mechanism in place to ensure model correctness; one such mechanism is constraints.

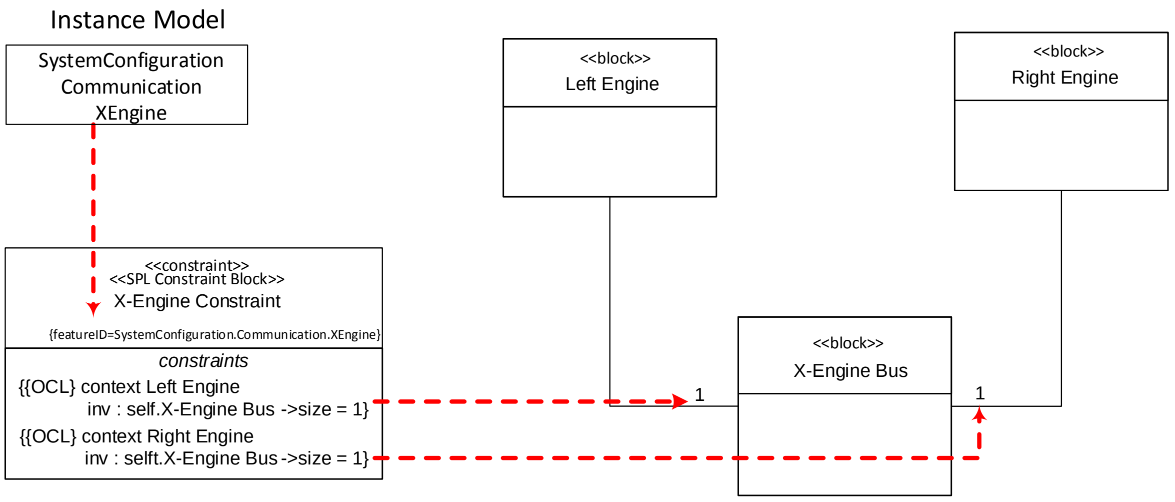
   A constraint is a statement that specifies invariant conditions that must hold for the system being modeled [31]. The idea is to specify the expected state of the model when a feature is selected as a list of constraints. Each of them will be stored in a constraint block, and linked to the controlling feature via its ID; they will execute only when an instance enables that particular feature.

   The example in figure 4.2a displays how the architecture model defines a constraint block with individual constraints that together will verify the consistency of the

[1]We assume here that a single feature applies to a requirement; this can be always archived by adding additional features as necessary. An alternative approach is to associate requirements with presence conditions, which are Boolean expressions over features.



(a) Example of constraints defined at the family architecture model.



(b) Example of constraints verifying the consistency of the model at the instance architecture model.

Figure 4.2: Variable Property Pattern.

system. In this case, the cardinality of X-Engine Bus must be one if the feature is present, or zero otherwise. This verification will happen only on instances that have the cross-engine communication enabled.

Note that to keep traceability simple, constraint blocks verify only if an individual feature is present in an instance. More intricate scenarios that require several comparisons at once must be handled directly by the FM, and exposed to the constraints as a single feature.

4.3.2 Instance Architectural Model

This section describes the steps that are required to create the instance architectural model. As with the family model, the process is iterative and can be executed as many times as required.

1. **Create Instance Model**
   The focus of this stage is to scope the behavior and structure of the instance being modeled. This process is typically accomplished by selecting some features from the FM, and storing them in an instance model.

2. **Retrieve and refine requirements**
   This stage focuses on determining the set of requirements that apply to the current instance; the ones that describe new features will be created from scratch, while there will be others carried over from the family architecture model. As described in section 3, requirements of this last type require different handling depending on their classification. One important remark is that this refinement process must be performed at both system and software levels.

3. **Model Structure and Behavior**
   This stage consists in customizing the family architecture model so that it fits the instance requirements. This process entails both creating new parts exclusive to the current instance and customizing the existing components. SysML enables this kind of restructuring via subclassing and redefinition; blocks can be modified by creating subclasses, and linking them with new components and properties via redefinition. Note that the top-level component, representing the system, will have one subclass for each instance; that block will represent the context of the new instance. This process is described in further detail in the following chapters.

4. **Verify Model Consistency**
   The instance must follow the conditions and restrictions stored in the constraint blocks at the family architecture model. Designers can leverage this information in two ways. First, one can get the applicable constraints based on the feature selection, and use them to verify consistency; any deviation from the expected state must be reported as a violation to fix. Second, one can use constraints as a guideline of the necessary changes to the model, which can then be applied via model transformation; this process is much more complicated. However, regardless of the approach, tools can be invaluable to streamline the amount of work necessary at this stage.

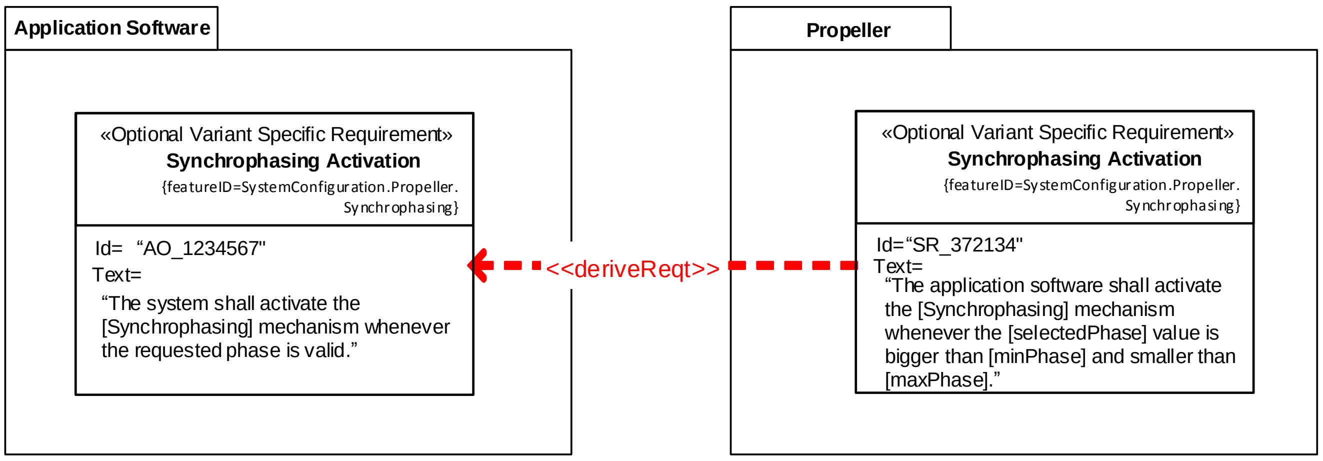
   In figure 4.2b, the constraint block *X-Engine Constraint* verifies that the cardinality of *X-Engine Bus* on both connectors is one. This check happens because the feature *XEngine* is activated in that instance.

5. **Model Export**
   Finally, at this point the instance model is ready to be used to remove the variability from the family architecture model; the result of this process allows different uses depending on the needs. One possible way is to import the model into an implementation tool, which will use it to produce actual code. Another potential use is to generate documentation; this is particularly useful in avionics, where the whole development process must be reported to certification authorities in several documents. Tool support is crucial at this stage, as it can significantly reduce manual work.

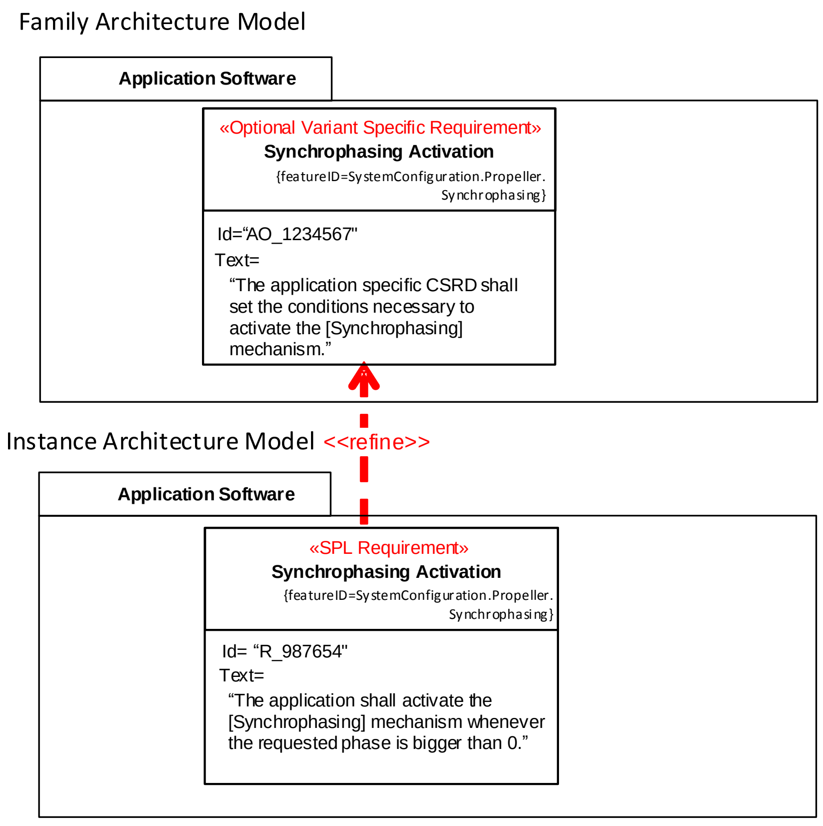
4.3.3 Traceability

As mentioned before, maintaining traceability is of utmost importance in avionics systems. To accomplish this goal, the model must contain connections linking all its elements. The first one occurs between the feature model and the requirements, which can be represented using the *«SPL Requirement»* stereotype (or one of its subclasses) and setting the corresponding feature ID.

The second type of links happens between requirements, and also with other parts of the model, like blocks. SysML contains a construct called dependency that is adequate to convey this kind of information. In fact, there are many of them, each of which is useful for different situations; table 4.1 provides more details. Figures 4.3 and 4.4 show examples of how traceability is used in practice.


Application Software
«Optional Variant Specific Requirement»
Synchrophasing Activation
{featureID=SystemConfiguration.Propeller.Synchrophasing}
Id= "AO_1234567"
Text=
"The system shall activate the [Synchrophasing] mechanism whenever the requested phase is valid."
<<deriveReqt>>
Propeller
«Optional Variant Specific Requirement»
Synchrophasing Activation
{featureID=SystemConfiguration.Propeller.Synchrophasing}
Id="SR_372134"
Text=
"The application software shall activate the [Synchrophasing] mechanism whenever the [selectedPhase] value is bigger than [minPhase] and smaller than [maxPhase]."

(a) System - Software Traceability.


Family Architecture Model
Application Software
«Optional Variant Specific Requirement»
Synchrophasing Activation
{featureID=SystemConfiguration.Propeller.Synchrophasing}
Id="AO_1234567"
Text=
"The application specific CSRD shall set the conditions necessary to activate the [Synchrophasing] mechanism."
Instance Architecture Model
<<refine>>
Application Software
«SPL Requirement»
Synchrophasing Activation
{featureID=SystemConfiguration.Propeller.Synchrophasing}
Id= "R_987654"
Text=
"The application shall activate the [Synchrophasing] mechanism whenever the requested phase is bigger than 0."

(b) Family Architecture Model - Instance Architecture Model Traceability

Figure 4.3: Traceability Examples.

Traceability	Relationship Keyword	Description
System - Software	«deriveReqt»	Provides a link between the top-level and lower-level requirements. It is particularly useful to depict the relationship between system and software, but can be used for intermediate traceability too.
Family - Instance	«refine»	Necessary for variant specific requirements. Highlights that extra information will be provided to eliminate, or at least, reduce the variability presented at the family model.
Requirement - Software Block	«satisfy»	Shows the software block(s) necessary to fulfill a requirement.
Requirement - Test Block	«verify»	Establishes a link with the tests that verify that the requirement has been satisfied indeed.

Table 4.1: SysML relationships to provide traceability.

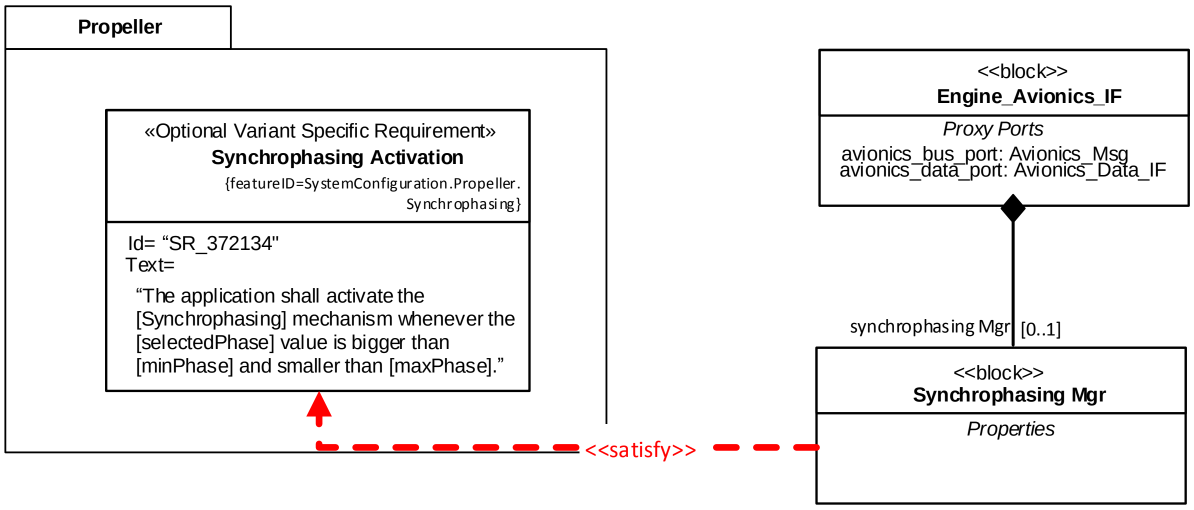

Propeller
«Optional Variant Specific Requirement»
Synchrophasing Activation
{featureID=SystemConfiguration.Propeller.Synchrophasing}
Id= "SR_372134"
Text=
"The application shall activate the [Synchrophasing] mechanism whenever the [selectedPhase] value is bigger than [minPhase] and smaller than [maxPhase]."
<<satisfy>>
<<block>>
Engine_Avionics_IF
Proxy Ports
avionics_bus_port: Avionics_Msg
avionics_data_port: Avionics_Data_IF
synchrophasing Mgr
[0..1]
<<block>>
Synchrophasing Mgr
Properties

Figure 4.4: Instance Architecture Model - Block Traceability.

4.4 Summary

This chapter introduced the proposed methodology to describe and model variability in avionics system product lines. First, it described the basic goals that guided the creation of the methodology, such as pervasive traceability, ease of use, and maintainability. Second, it explained how it relates to previous work and what the new contributions are. Third, it provided a sequence of steps necessary to create each of the models; the family and the instance architecture models. Finally, it provided suggestions about how to provide traceability with existing SysML relationships and how to keep the models consistent via constraints.

Chapter 5

Behavioral Patterns

As mentioned before, behavioral diagrams describe a system in terms of functionality; they focus more on how the system must behave than on how it is structured. This information can be important as it can aid in choosing the decomposition of the system. From a variability point of view, understanding how behavior can change between instances can allow organizing components in a way that will minimize variations and their impact on the system. This chapter describes how alternative behavior can arise in activity and sequence diagrams and proposes two patterns to deal with them.

5.1 Variability in Activity Diagrams

5.1.1 Scenario Description

One common way to describe the operation of a system is by explaining how the provided inputs are processed and transformed into some expected output. The actual nature of these elements can differ; sometimes they will be concrete objects, like flows of fuel or air, while others will be more abstract, like data or signals. What is important is that they are transformed through an ordered execution of steps. In SysML, the best way to express this interaction is through activity diagrams; they are the primary representation for modeling flow-based behavior.

Although family architecture models can also benefit from flow-based behavior descriptions, creating activity diagrams for them is not straightforward. For these diagrams to be useful, they have to show what parts of the behavior are common to all products, and which ones change between variants. This representation shall fulfill the following goals:

- There must be a precise demarcation of where the differences in behavior start and end.
- The diagram is required to show clearly which features control the variance in behavior.
- There should be enough information to determine which inputs and outputs will be present depending on the feature selection.

5.1.2 Suggested Pattern

SysML activity diagrams have conditional nodes, which represent an exclusive choice among some number of alternatives [45]. All conditional nodes contain one or more clauses, each with a test and body section. The body of a clause will execute only if its corresponding test yields true; at most one body clause can be executed within any conditional node. This SysML construct can be leveraged to display variability by extending it via stereotypes. The idea is to tag each of them with the ID of the feature whose presence can modify the behavior of the activity, and classify the node as «optional»or «alternative», depending on the possible values that the feature can have. Nodes tagged as optional will only have one clause while alternative ones can have two or more; the body parts hold the functionality to execute when the feature is enabled.

A difference in behavior can also affect the parameters that the activity needs or will produce. To make the interface as explicit as possible, mark inputs and outputs that are not always present with multiplicity [0..1], or with the optional stereotype; these two have the same semantics, and are already part of the SysML specification, so either one can be used.

Finally, there can be cases where a piece of behavior will always be different for each variant. When that happens, that functionality can be encapsulated into one action, and then marked with the stereotype «customization point». With that tagging, the model specifies that all variants must redefine the action with their custom logic.

In summary, these are the guidelines to show variability in activity diagrams:

1. Show alternative behavior in conditional nodes; tag them with the corresponding feature ID.
2. Mark parameters not needed by all products with multiplicity [0..1] or the «optional»stereotype.

3. Mark actions that variants must redefine with the stereotype «customization point».

Example

The pitch of the blades of a propeller can be managed via a solenoid; adding or removing pressure from it will alter the position of the blades. The controller can make use of this mechanism to regulate the propeller speed, which in turn will help modify the thrust of the whole engine. However, determining the right pressure to command to the solenoid can vary depending on the equipment in the engine. If it contains a sensor to measure the current blade angle, the value can be controlled in a closed-loop. Otherwise, it must be regulated indirectly in an open-loop.

Figure 5.1 shows this scenario in an activity diagram. If the *Closed_Loop* feature is enabled, the engine can just calculate the right pressure to use. On the other hand, if the *Open_Loop* feature is selected, the controller must request a default pressure delta until the propeller reaches the desired speed. Note that the variability is contained within a conditional node, which is tagged with the ID of the parent feature. Furthermore, each clause will execute only if the right child of that feature is enabled. If *Open_Loop* is selected then its body will execute. Otherwise, the lower one will be used.

Discussion

There are multiple benefits of modeling variability with this pattern. First of all, it becomes clear which behavior can change, where these differences start and end, and which features control them. Additionally, changes to the interface are made explicit; this will help define the data flows in the structural diagrams. Furthermore, the effort of making such a diagram is not wasted. Modeling early in the development stages can help understand variability in the system better, reduce ambiguity, and the model can be reused later for documentation purposes.

A potential disadvantage of using this variability representation is that the allocation of behavior to structural elements via swimlanes might not be possible anymore; their combination could make it ambiguous to determine which part of the functionality belongs to a particular block. Therefore, with this pattern, allocation would always need to be specified via other means, like callout or matrix notation.

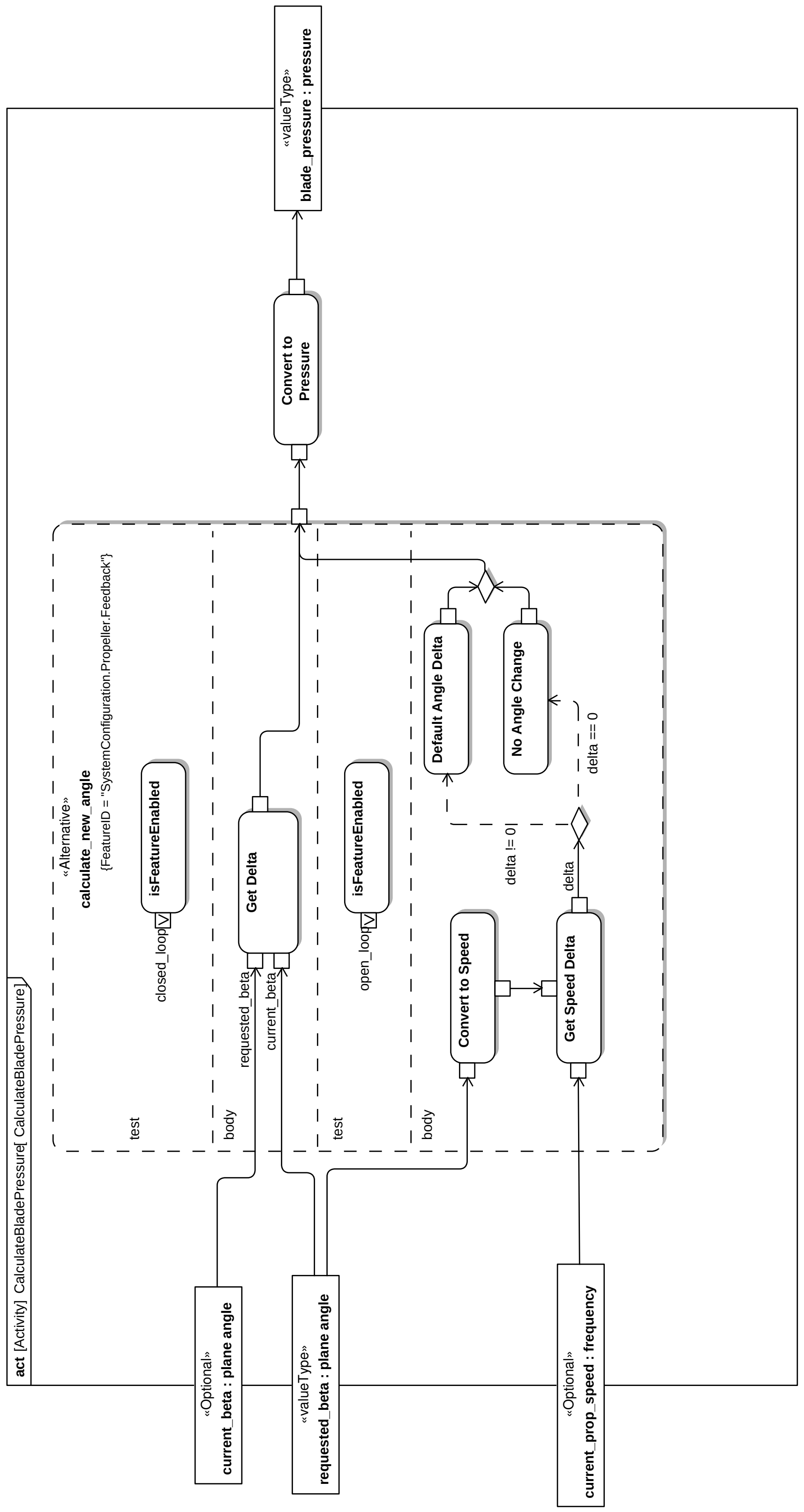


Figure 5.1: Variability in Activity Diagram

5.2 Variability in Sequence Diagrams

5.2.1 Scenario Description

One way to understand the behavior of a system is by identifying how its different components react to internal and external stimulus. This data is quite important, since such interactions can become quite complex in big systems. SysML provides sequence diagrams for this purpose; they describe the flow of control between actors and system (blocks) or between parts of a system [32].

In a family architectural model, showing the variability that arises in these interactions can be very helpful. This information can aid in the identification of the components that are necessary by each product in the family to fulfill its operational requirements. To achieve that this variability representation must meet the following goals:

- There must be a precise demarcation of where the differences in behavior start and end.
- The diagram is required to show clearly which features control the variance in behavior.
- There should be enough information to determine which actors and components will be part of the interaction in each variant.

5.2.2 Suggested Pattern

This pattern takes advantage of a construct in sequence diagrams called combined fragments. There are many types of them, each specifying different semantics to the structure of the messages that it contains. As described in [21], a combined fragment consists of an interaction operator and its operands. The interaction operator defines the type of message structuring, and its operands are all subject to that rule. Furthermore, each operand has a guard with a constraint that indicates the conditions under which it is valid.

There are two operators that are particularly useful for variability modeling: *opt* and *alt*. The first one specifies that the operand of the fragment will get executed only if the guard evaluates to true. The latter works similarly; the only difference is that instead of one, the fragment contains multiple operands, out of which only one will be executed depending on the value of the guard. The idea is to place variable interaction within combined fragments *opt* or *alt*, depending on the type of variability. Then each fragment is marked with the «SPL fragment»stereotype and tagged with the ID of the controlling feature; each operand will execute only if that ID is selected.

One important aspect to highlight is that the guard of the operator of the combined fragment is bound to a single lifeline and can reference only attributes of that lifeline in its constraint. However, the determination of which operand to execute is controlled by the feature ID, specified in the «SPL fragment»stereotype. For this pattern to work, the modeling tool must be able to access this value.

If an actor or component lifeline exists only within an operand of a combined fragment, it will not be included on the variants that do not select the feature that controls that optional interaction.

In sum, variability in sequence diagrams can be implemented in the following way:

1. Use combined fragments to specify changing behavior; use *alt* interaction operator to show alternative behavior and *opt* for optional functionality.

2. Tag these fragments with «SPL fragment»stereotype, to show that they are controlled by the FM, and specify the ID of the controlling feature.

Example

Typically, the speed of the propeller in an engine is determined by the power level selected by the pilot. However, aircrafts with multi-engine installations might have a feature called *synchrophasing* to reduce noise and vibration, which might also affect the speed of the propeller.

Figure 5.2 shows an example of a system with this choice. In it, part of the speed determination process is enclosed within an *opt* combined fragment. This part of the model will execute only if the *synchrophasing* feature is enabled.

Discussion

As in activity diagrams, modeling variability using this pattern allows displaying where changes in behavior can happen, what is the span of the differences, and which features control them. Also, it can help visualize how internal components interact with each other, and with external systems and actors, depending on the features of the variant. All this information is helpful for documentation purposes and will be quite valuable for defining structural models.

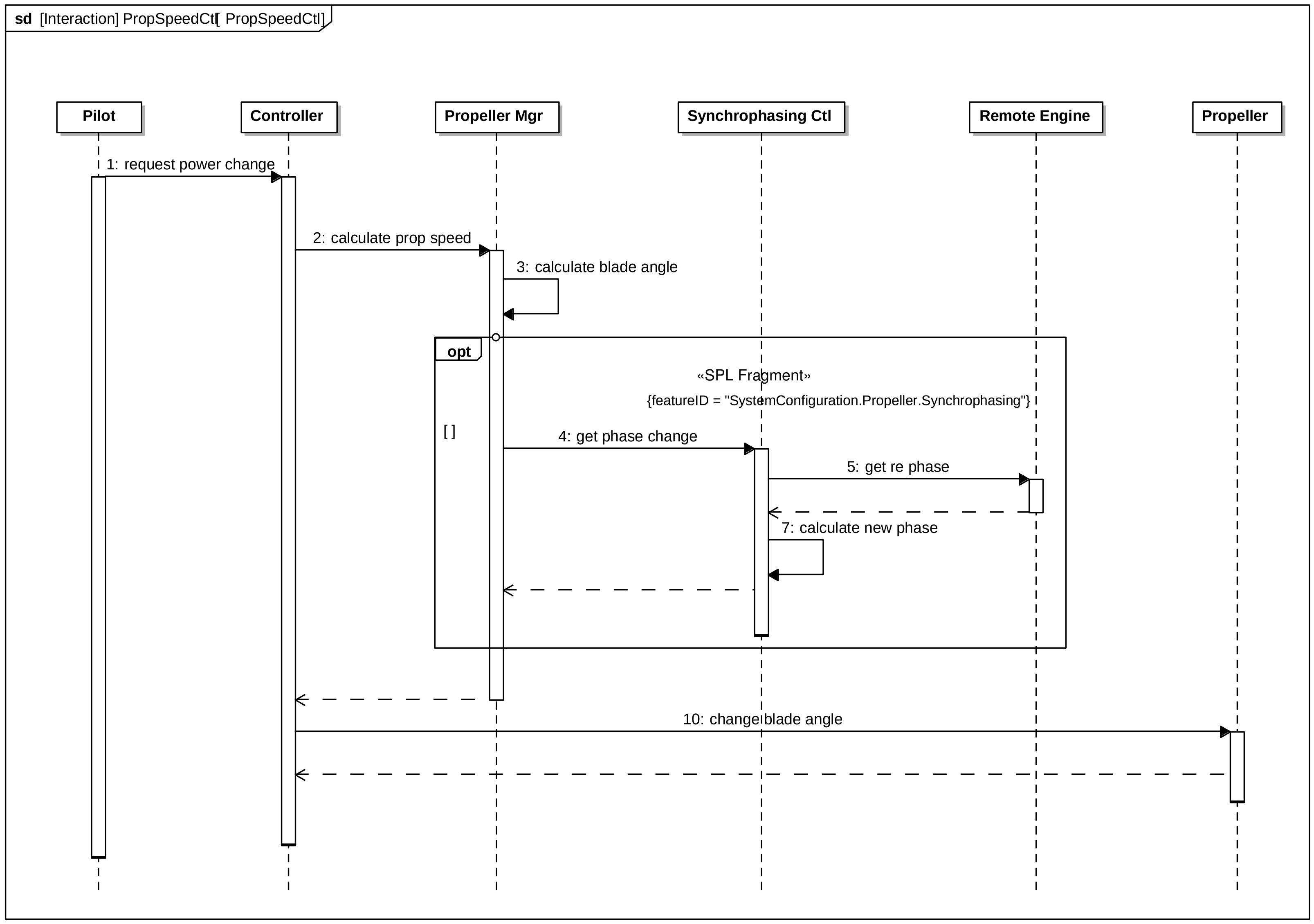


Figure 5.2: Variability in Sequence Diagram

5.3 Summary

This chapter described how variability presents in two behavioral diagrams: activity and sequence diagrams. In the first, variations can occur in the interfaces and on the sequence of steps necessary to transform the provided inputs into the expected outputs. This chapter proposed making use of conditional nodes and the *optional* stereotype to deal with the differences. In the second, interactions between lifelines can change between instances. The chapter suggested using combined fragments to show the message variations.

Chapter 6

Structural Patterns

Structural diagrams in SysML describe the arrangement of the system. In particular, they specify the components that make up the system, how they connect to each other, and the flows they contain; all this information is stored in block definition (BDD) and internal block diagrams (IBD). This chapter describes how variability arises and formulates a series of patterns to keep this variation under control; this is important as it can help reduce the amount of work necessary to create and maintain diagrams with variability.

6.1 Variable Property

6.1.1 Scenario Description

Differences in the model are not restricted to structural or behavioral variability. There are situations where the components that make up a system stay the same, but the values of one or more value properties [1] change from instance to instance. This case can be referred to as quantitative variability and can manifest itself in two ways: a change in value or range (valid interval of values).

Systems can have hundreds or even thousands of value properties, so keeping control of which ones change between variants can be challenging. Models can be helpful to convey this information in a structured way. Any approach used to model quantitative variability must have the following characteristics:

[1] In SysML, a value property is a quantitative characteristic of a block such as weight or speed [21].

- It must display what value properties are subject to change between variants.
- The model must show what approach is taken to deal with the changes.
- Model impact should be as small as possible; only the block with the variable property shall change. The rest of the components shall be left untouched.

6.1.2 Suggested Pattern

Depending on the particular situation, quantitative variability can require different handling. For this reason, this pattern defines three ways to deal with it. The first one handles the case where only the value of the value property changes between instances. When this happens the family architecture model does not require any special treatment. Blocks only need to include the value properties as normal along with their default values, if necessary. Then, instance models that wish to change them only have to subclass the top level block, and assign the new values via an IBD; SysML calls this data context-specific values since it is only valid within the scope of the current instance.

The second way deals with cases where the valid range of possible values requires a change. Although it would be very practical to use IBD diagrams for this purpose too, SysML currently disallows specifying intervals as context-specific values. Therefore, this kind of variability demands a different type of approach; it is necessary to subclass the block, redefine the corresponding value property, and specify the new interval.

The final way covered by this pattern is quite different. It manages cases where the actual values are necessary by the system, but they are not relevant or does not make sense to specify them while modeling. Maybe the data varies too often, and the actual values do not matter to the model as long as they are within adequate ranges. For this situation, it is necessary to mark the properties with the «PDI »stereotype[2]. The intent is to highlight that the data must be provided at some point during the system lifecycle but that it is currently unknown.

Adding everything up, these are the steps necessary for this pattern:

Value Properties

1. Model family architecture without adding any extra information to the property whose value might change.

[2]PDI stands for Parameter Data Item, as defined by DO178C [13]

2. For instance models, subclass top-level block. This step is necessary for all patterns.
3. Use IBD to provide new values.

Intervals

1. Subclass block with value property to change.
2. Redefine corresponding value property.
3. Provide new interval.

PDI

1. Mark properties with values that need to be provided in the future with the «PDI »stereotype.
2. Provide the concrete data at some point during the system lifecycle.

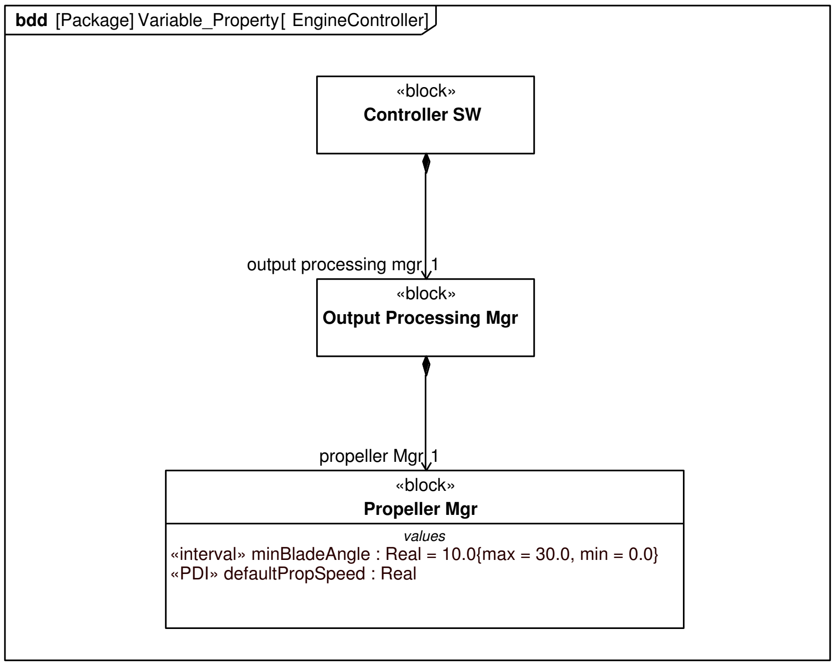
Example

Figure 6.1a shows a part of the *Engine Controller* BDD. It includes the *Propeller Manager*, which is the component in charge of controlling the operation of the propeller in a turboprop engine, and the blocks that include it. The value property *minBladeAngle* is the one whose value is prone to change on each variant. Figure 6.1b shows how to modify them by creating context-specific values in an IBD; the compartment marked as *defaultValue* contains these new values. Conversely, figure 6.1c contains a BDD that displays how to modify the interval. In this case, the whole *Propeller Manager* needs to be subclassed, and the value property is redefined to be able to access and modify the corresponding interval.

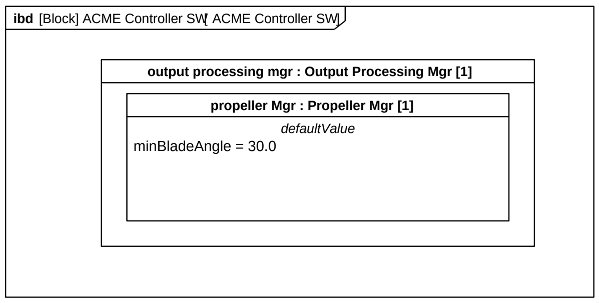
Also, *Propeller Manager* contains the *defaultPropSpeed* value property, marked with the «PDI »stereotype. This tagging means that the exact value of the property is not important at this point, but it will be necessary at some point in the future.

Discussion

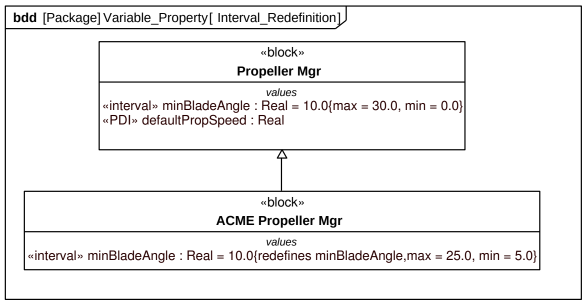
Defining context-specific values to handle quantitative variability is very easy and does not have any model side effects. However, it is possible that most instances will require very little or no context-specific values defined while modeling. Probably, it will be more


bdd [Package] Variable_Property[EngineController]
«block»
Controller SW
output processing mgr 1
«block»
Output Processing Mgr
propeller Mgr 1
«block»
Propeller Mgr
values
«interval» minBladeAngle : Real = 10.0{max = 30.0, min = 0.0}
«PDI» defaultPropSpeed : Real

(a) Engine Controller BDD


ibd [Block] ACME Controller SW[ACME Controller SW]
output processing mgr : Output Processing Mgr [1]
propeller Mgr : Propeller Mgr [1]
defaultValue
minBladeAngle = 30.0

(b) IBD of the *ACME Controller SW* block, which shows how to assign context-specific values.


bdd [Package] Variable_Property[Interval_Redefinition]
«block»
Propeller Mgr
values
«interval» minBladeAngle : Real = 10.0{max = 30.0, min = 0.0}
«PDI» defaultPropSpeed : Real
«block»
ACME Propeller Mgr
values
«interval» minBladeAngle : Real = 10.0{redefines minBladeAngle,max = 25.0, min = 5.0}

(c) BDD that shows interval redefinition.

Figure 6.1: Variable Property Pattern.

important to keep track of the valid intervals for each instance as that information could be invaluable to verify the correct state of the system.

Also, it might not be evident, but marking value properties with the «PDI »stereotype can be of great use; a tool could mine those tags and produce a report of all the values required by an instance depending on its features. This approach could be particularly attractive if the model produces only one executable, and the PDIs are used to configure variants as suggested by DO178C [13].

6.2 Optional Component

6.2.1 Scenario Description

A very common type of structural variability happens when the status of a feature can trigger the addition or complete removal of a component in the system. Any modeling strategy used to solve this problem must meet the following goals:

- Regardless of the status of the feature, the model shall always be in a consistent state.
- If desired, it should be possible to remove unused code; this could be very useful to prevent accidental activation[3].
- There should not be any unnecessary conditional statements scattered through the code checking the status of the feature.

6.2.2 Suggested Pattern

There are two proposed ways to handle this scenario. Although both rely on the same basic concepts (inheritance and cardinality management), they differ on the impact they have on the rest of the model. Each approach has pros and cons, which need to be weighed depending on the concrete model.

[3]Accidental activation used to be a major concern during the development of avionics systems. There was always the risk that a high-energy particle would leave the code in an inconsistent state, which could lead to dangerous situations. However, current hardware has addressed this problem for the most part.

Suggested Pattern 1: Component removal

The most straightforward way to model an optional component at the family architecture level is through a composite aggregation relationship between the component itself and the block that owns it. The cardinality at the part end-point is set to [0..1]. Each instance of the family will then subclass the aggregate block and redefine this cardinality; it needs to be set to one if the block is present, or zero otherwise.

When a component is marked as optional by setting its cardinality to zero, all its input and output is not required anymore and might need to be removed. This change has to be reflected on the aggregate block to avoid any unnecessary data from flowing through the system. Commonly, this entails subclassing the interface block of the input/output ports and modifying their cardinalities accordingly. This process is described in more detail in section 6.5.

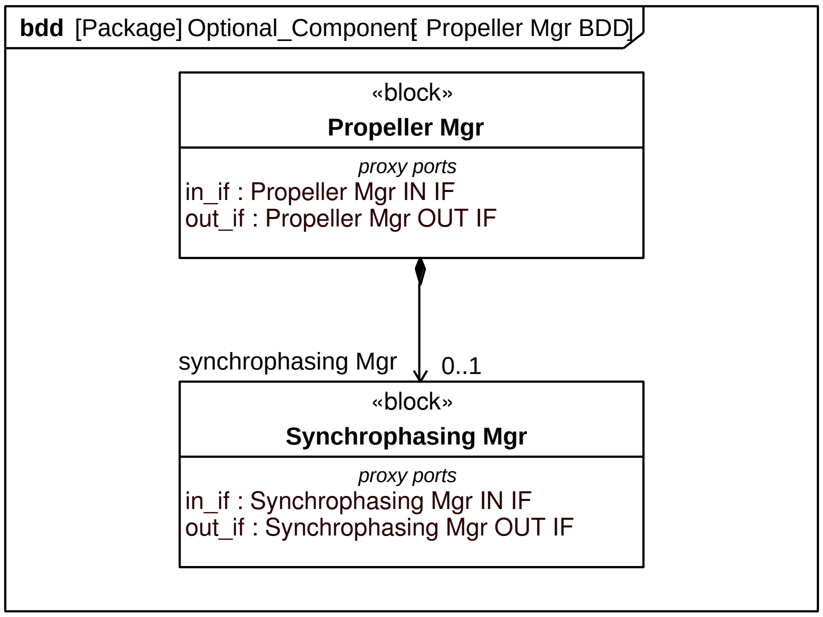
Adding it up, these are the steps necessary to use this pattern:

1. Subclass the block that contains the optional component and redefine the end-point cardinality to zero or one as necessary.
2. Update the flows of the aggregate block.
3. Keep both changes, redefining cardinality and updating flows, synchronized via constraints.

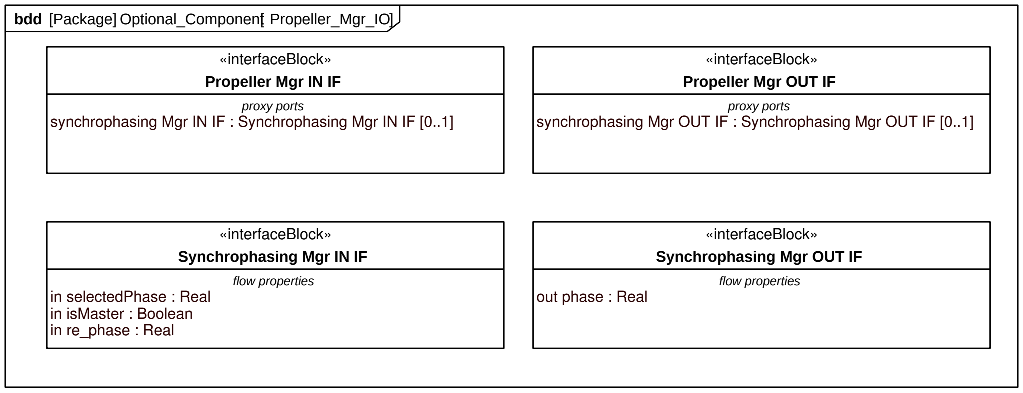
Example

When two or more turbo-prop engines operate in an aircraft at the same time, they produce noise and vibration, which can become unpleasant if they are running at slightly different speeds. To minimize this nuisance the controller system can match the phase and rotational speed of the propellers; this is called synchrophasing. As this synchronization is required only in multi-engine installations, the corresponding feature is marked as optional.

The example in figure 6.2 contains part of the *Propeller Mgr BDD*. It shows the blocks required to enable propeller synchrophasing. Since this feature is optional, the block *Synchrophasing Mgr*, which manages this functionality, has cardinality [0..1]. Also, notice that the corresponding flows in the interface block are also optional; both of these cardinalities must be in synch. Figure 6.3, on the other hand, shows the BDD of an instance that does not enable synchrophasing.

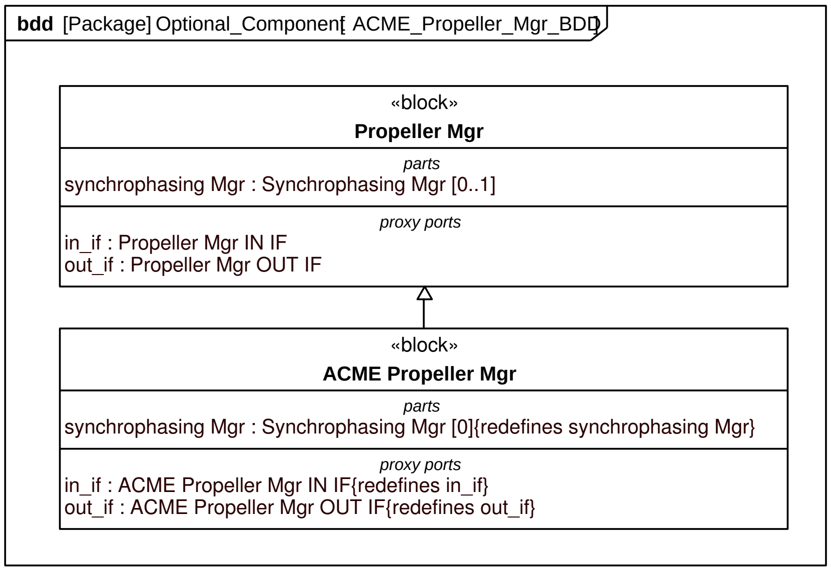

bdd [Package] Optional_Component[Propeller Mgr BDD]
«block»
Propeller Mgr
proxy ports
in_if : Propeller Mgr IN IF
out_if : Propeller Mgr OUT IF
synchrophasing Mgr
0..1
«block»
Synchrophasing Mgr
proxy ports
in_if : Synchrophasing Mgr IN IF
out_if : Synchrophasing Mgr OUT IF

(a) Propeller Manager BDD

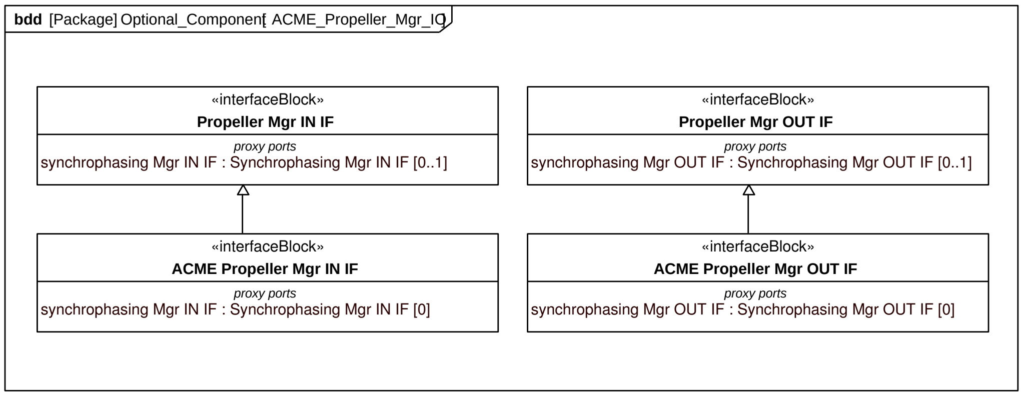


(b) Interfaces used by the *Propeller Mgr* and *Synchrophasing Mgr* blocks.

Figure 6.2: Component Removal Pattern. Family Architecture level.


bdd [Package] Optional_Component[ACME_Propeller_Mgr_BDD]
«block»
Propeller Mgr
parts
synchrophasing Mgr : Synchrophasing Mgr [0..1]
proxy ports
in_if : Propeller Mgr IN IF
out_if : Propeller Mgr OUT IF
«block»
ACME Propeller Mgr
parts
synchrophasing Mgr : Synchrophasing Mgr [0]{redefines synchrophasing Mgr}
proxy ports
in_if : ACME Propeller Mgr IN IF{redefines in_if}
out_if : ACME Propeller Mgr OUT IF{redefines out_if}

(a) BDD of instance without synchrophasing.


bdd [Package] Optional_Component[ACME_Propeller_Mgr_IO]
«interfaceBlock»
Propeller Mgr IN IF
proxy ports
synchrophasing Mgr IN IF : Synchrophasing Mgr IN IF [0..1]
«interfaceBlock»
Propeller Mgr OUT IF
proxy ports
synchrophasing Mgr OUT IF : Synchrophasing Mgr OUT IF [0..1]
«interfaceBlock»
ACME Propeller Mgr IN IF
proxy ports
synchrophasing Mgr IN IF : Synchrophasing Mgr IN IF [0]
«interfaceBlock»
ACME Propeller Mgr OUT IF
proxy ports
synchrophasing Mgr OUT IF : Synchrophasing Mgr OUT IF [0]

(b) Interfaces used by the instance without synchrophasing.

Figure 6.3: Component Removal Pattern. Instance level.

Discussion

Using this pattern allows the addition or removal of optional parts as necessary and adapts the whole model to accommodate the change. The biggest benefit is that the resulting model is very clean; instances do not contain unnecessary components, and their interfaces will include only the required input and output flows.

However, the pattern entails keeping several parts of the model in sync, which might require more manual work or tool support. Moreover, using this approach makes sense only if the actual implementation will mirror it: removing the optional component altogether, adapting interfaces (inputs/outputs), and modifying the rest of the code to address the change.

Suggested Pattern 2: Mock component

An alternative way to model an optional component is by creating an abstract block that will be subclassed and replaced in an instance by one of two possible substitutes. The first one contains the real structure and behavior, and must be used whenever the optional feature is enabled. In contrast, the second one is just a mock (stub). It is an empty placeholder that ignores all inputs and only returns default values. It shall be used if the optional feature is disabled.

These are the steps required to use this pattern:

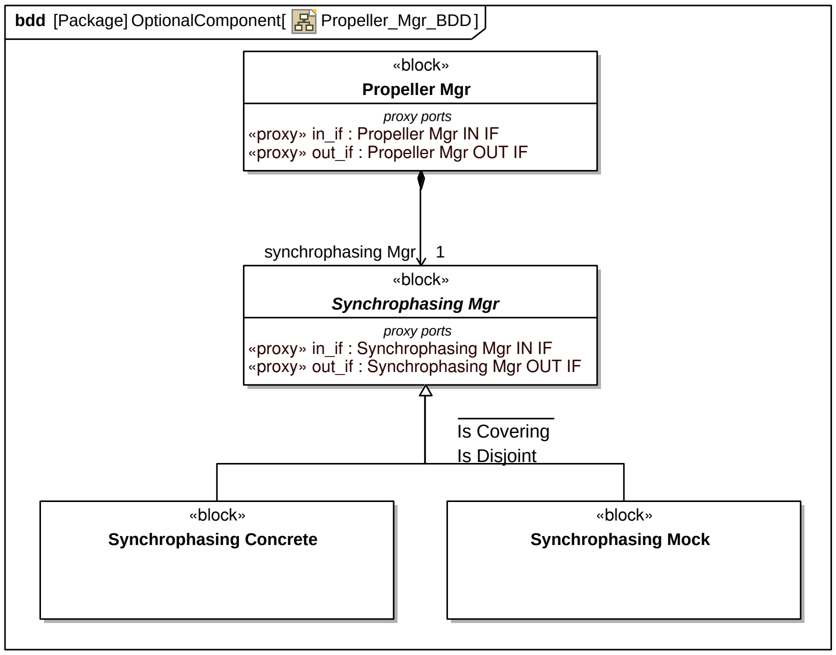
1. The family architecture model includes a block that represents the optional component. It must be an abstract block with two subclasses.

2. The instance model must subclass the aggregate block, and redefine the type of the optional component with either the real or mock block.

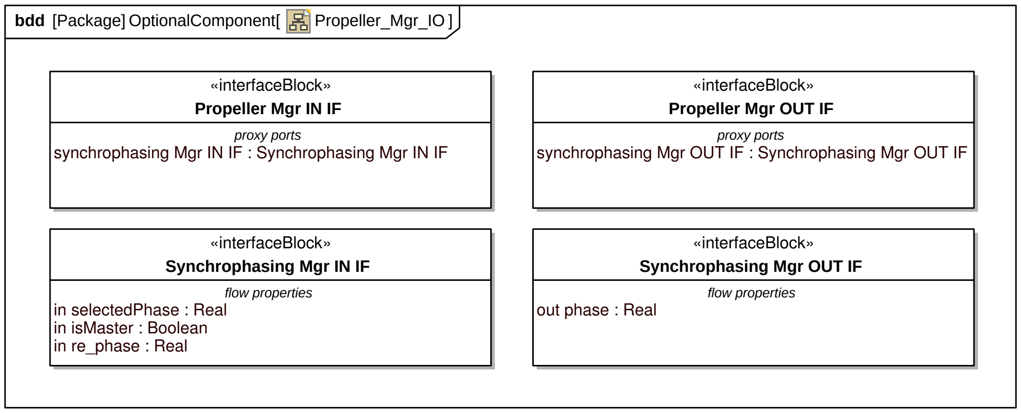
3. No changes to interface blocks are necessary, since the inputs/outputs are the same regardless of the chosen block.

Example

Figure 6.4 presents the same synchrophasing example as in section 6.2.2, but instead of removing the block it makes use of the mock component pattern. In 6.4a, the BDD of the family architecture model contains the abstract block *Synchrophasing Mgr*, which is subclassed by *Synchrophasing Concrete* and *Synchrophasing Mock*; the former represents

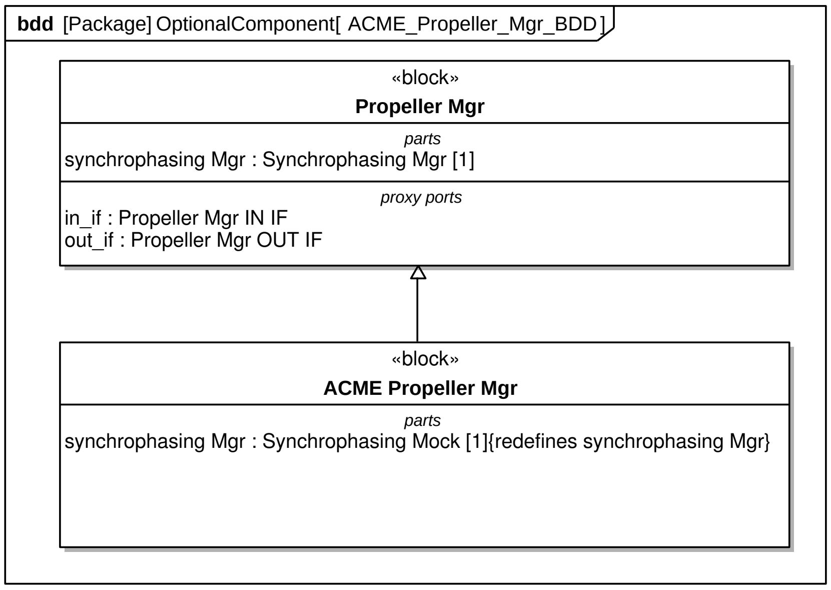

bdd [Package] OptionalComponent[Propeller_Mgr_BDD]
«block»
Propeller Mgr
proxy ports
«proxy» in_if : Propeller Mgr IN IF
«proxy» out_if : Propeller Mgr OUT IF
synchrophasing Mgr
1
«block»
Synchrophasing Mgr
proxy ports
«proxy» in_if : Synchrophasing Mgr IN IF
«proxy» out_if : Synchrophasing Mgr OUT IF
Is Covering
Is Disjoint
«block»
Synchrophasing Concrete
«block»
Synchrophasing Mock

(a) Propeller Manager BDD



(b) Interfaces used by the *Propeller Mgr* and *Synchrophasing Mgr* blocks.

Figure 6.4: Mock Pattern. Family Architecture level.


bdd [Package] OptionalComponent[ACME_Propeller_Mgr_BDD]
«block»
Propeller Mgr
parts
synchrophasing Mgr : Synchrophasing Mgr [1]
proxy ports
in_if : Propeller Mgr IN IF
out_if : Propeller Mgr OUT IF
«block»
ACME Propeller Mgr
parts
synchrophasing Mgr : Synchrophasing Mock [1]{redefines synchrophasing Mgr}

(a) BDD of instance with mock component.

Figure 6.5: Mock Pattern. Instance level.

the real implementation, while the latter is the mock. Figure 6.5 shows an instance selecting the mock block since the feature is not enabled. Notice that no interface changes are necessary.

Discussion

This pattern allows enabling or disabling an optional component with minimum impact to the model. It is an excellent choice whenever it is not practical to update the implementation to accommodate the presence or absence of an optional block. For instance, it might be expensive or unfeasible for a company to test multiple versions of a component because of interface changes. In that case it is probably better to keep interfaces consistent, and just provide default values whenever the element is not present. The mock block is perfect for that design and to convey the default values.

The downside of the pattern is that the interfaces are not updated. Therefore, unnecessary data is kept flowing through the system.

6.3 Alternative Implementation (XOR)

6.3.1 Scenario Description

Variability in a system often arises as different possible ways of structuring code to provide alternative functionality based on feature selection. This kind of changes probably requires swapping some components and adapting the rest of the system. Any strategy used to model this scenario must satisfy the following objectives:

- The model shall allow selecting only one of the possible alternatives.
- The model must always be in a consistent state.
- If required, it shall be possible to remove any unused code; this could be useful to reduce executable size and prevent any possible accidental activation.
- There should not be any unnecessary conditional statements scattered through the code checking the status of the feature.

6.3.2 Suggested Pattern

This pattern requires the family architecture to show what the possible variations are by leveraging inheritance. The idea is to include an abstract block to represent the variable component. If the alternatives are known at this time, they are included in the model as subclasses of the abstract block. Otherwise, the block is marked with the «customization point »stereotype, which will force each variant to subclass it with a new unique implementation.

Later, instances will create a child of the aggregate block, and replace the variable component with either one of the given options or their unique implementation. Note that even if the family model already offers explicit alternatives, the instances are free to create their own. If designers want to prevent this from happening, they can tag the generalization relationship with generalization sets specifying that all the possible alternatives have been given already.

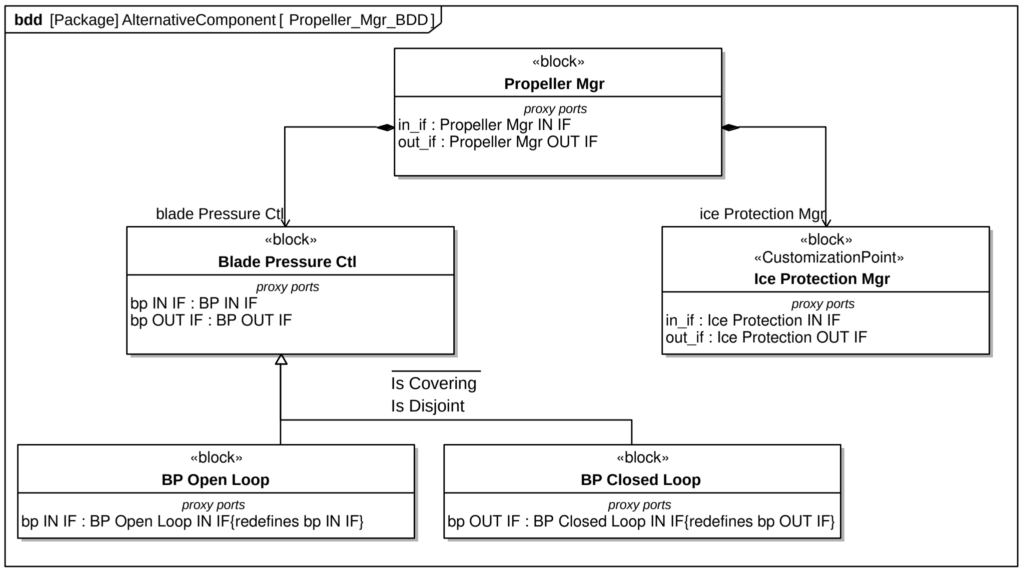
It is important to mention that alternative components can modify their interface if necessary. To do so, they need to redefine the type of the corresponding proxy port by using redefinition. That way they can specify new inputs and outputs as required. Obviously, this implies that other parts of the model must be in synch with these changes.

As a summary, the steps required to use this pattern are:

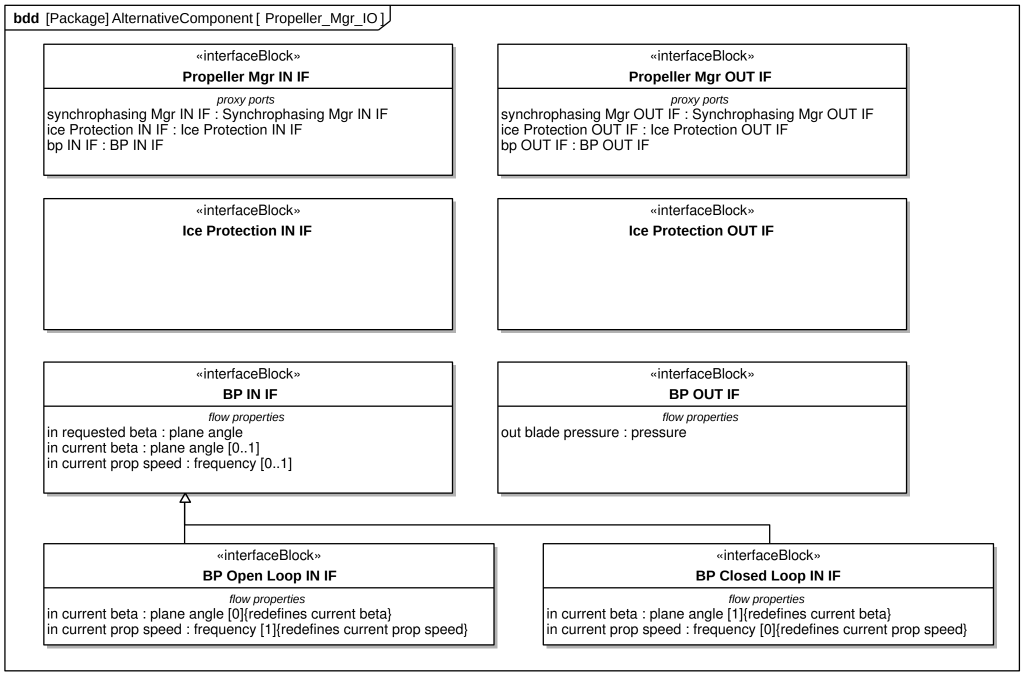
1. The family architecture model includes an abstract block to represent the variable component. If the alternatives are known at this time, they are shown as subclasses of this block. Otherwise, the abstract block is marked with the «customization point»stereotype.
2. The instance model subclasses the aggregate block and redefines the type of the variable component to one of the given alternatives or to the one created exclusively for the current instance.
3. If the alternatives have different interfaces, they must create new interface blocks with the new signature and redefine the corresponding proxy ports.

Example

In a turbo-prop, one of the ways a controller can modify the thrust produced is by adjusting the pitch of the propeller. Whenever the software receives a command to produce a particular power level, the controller measures the current position of the propeller blades,

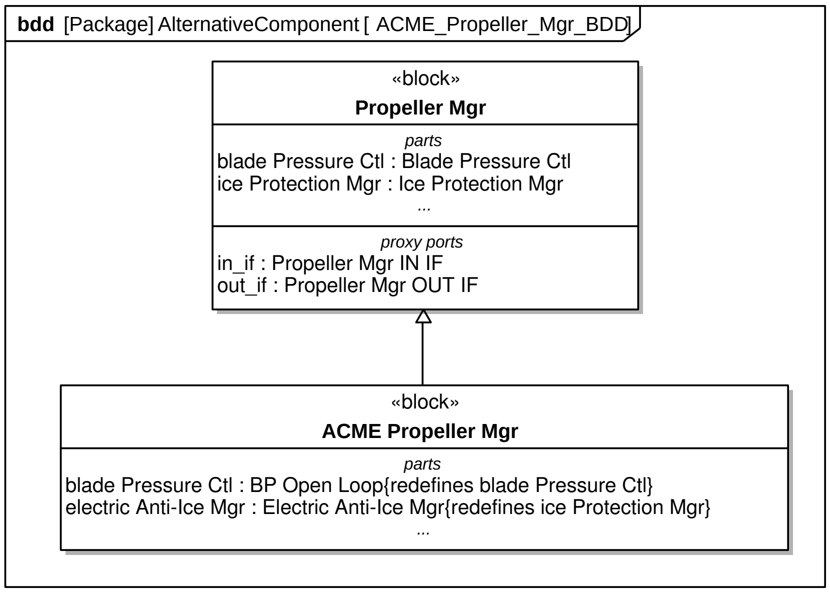


(a) Propeller Manager BDD


bdd [Package] AlternativeComponent [Propeller_Mgr_IO]
«interfaceBlock»
Propeller Mgr IN IF
proxy ports
synchrophasing Mgr IN IF : Synchrophasing Mgr IN IF
ice Protection IN IF : Ice Protection IN IF
bp IN IF : BP IN IF
«interfaceBlock»
Propeller Mgr OUT IF
proxy ports
synchrophasing Mgr OUT IF : Synchrophasing Mgr OUT IF
ice Protection OUT IF : Ice Protection OUT IF
bp OUT IF : BP OUT IF
«interfaceBlock»
Ice Protection IN IF
«interfaceBlock»
Ice Protection OUT IF
«interfaceBlock»
BP IN IF
flow properties
in requested beta : plane angle
in current beta : plane angle [0..1]
in current prop speed : frequency [0..1]
«interfaceBlock»
BP OUT IF
flow properties
out blade pressure : pressure
«interfaceBlock»
BP Open Loop IN IF
flow properties
in current beta : plane angle [0]{redefines current beta}
in current prop speed : frequency [1]{redefines current prop speed}
«interfaceBlock»
BP Closed Loop IN IF
flow properties
in current beta : plane angle [1]{redefines current beta}
in current prop speed : frequency [0]{redefines current prop speed}

(b) Interfaces used by the *Propeller Mgr*, *Blade Pressure Ctl* and *Ice Protection Mgr* blocks.

Figure 6.6: Alternative Component Pattern. Family Architecture level.


bdd [Package] AlternativeComponent [ACME_Propeller_Mgr_BDD]
«block»
Propeller Mgr
parts
blade Pressure Ctl : Blade Pressure Ctl
ice Protection Mgr : Ice Protection Mgr
...
proxy ports
in_if : Propeller Mgr IN IF
out_if : Propeller Mgr OUT IF
«block»
ACME Propeller Mgr
parts
blade Pressure Ctl : BP Open Loop{redefines blade Pressure Ctl}
electric Anti-Ice Mgr : Electric Anti-Ice Mgr{redefines ice Protection Mgr}
...

(a) BDD of instance with alternative components.

Figure 6.7: Alternative Component Pattern. Instance level.

calculates a new position, and requests the necessary adjustment. However, there are some engines that do not have the equipment required to get this measurement; they have to rely on other means, like the propeller speed, to control the system. Figure 6.6a contains the BDD that shows this variability. In this diagram, *Blade Pressure Ctl* is an abstract block that presents an alternative; it can be replaced by *BP Closed Loop* when an engine has the means to measure the current blade angle, or *BP Open Loop* when it cannot. As shown in figure 6.6b, each of these variants requires different interfaces, so their port types must be redefined too.

Another important feature of propellers is the way they can use to prevent from getting covered with ice as that accumulation could decrease their performance or, in the worst case, make the system lose control of the propeller. However, there are many types of mechanisms that can provide this feature. By using «Customization Point»stereotype, block *Ice Protection Mgr* in figure 6.6a forces each engine to have an ice protection system, but does not specify the possibilities; each concrete variant must define its own.

Discussion

Using this pattern allows variants to choose from different alternatives. If they do not change the abstract block interface, swapping components is completely transparent; no other parts are touched. However, if the interface does change, more updates might be necessary to keep the model consistent. In particular, blocks that produce or consume one of the changed flows might require to be adapted.

6.4 Collection

6.4.1 Scenario Description

Structural changes are not limited to individual components. There are cases where the number of elements of a particular type can vary depending on the feature model. For example, an aircraft can contain one or more engines. Models should be able to represent this type of variability efficiently; changes to the multiplicity of an element should not produce a noticeable impact on its structure. An adequate modeling strategy should accomplish the following objectives:

- It shall be possible to modify the actual number of elements of the same type without affecting the system structure.

- Each variable element can have alternative implementations (with some restrictions).
- Minimize the number of conditional statements checking the status of the feature.

6.4.2 Suggested Pattern

This pattern is very simple; instead of working with a variable number of independent elements, the model should organize them as a single collection. This approach allows the system to remain oblivious, up to a certain level, of the actual number of them. For it to work, all items must belong (or derive) from the same type, and have the same interface. When they do not meet these two conditions they require different handling; they do not make sense as a group.

At the family architecture model, a component is shown to have a collection with a variable number of items via a containment relationship, whose multiplicity must have a lower bound of at least 1 and an upper bound of at least 2. This range restricts the possible amount of elements that any instance can have.

Later on, variant models have to provide extra information about the collection. At the very least, they have to subclass the aggregate block and specify the concrete number of items that it holds by redefining the property and changing its cardinality. If necessary, the model can also provide additional information about one or more of the elements via subsetting.

One important restriction to keep in mind is that this pattern requires the collection cardinality always to be greater than zero. Otherwise, there is the possibility of having an instance without any element, which would mean that the group of items as a whole is not always present; this case requires the optional component pattern.

Again, these are the steps required to use the collection pattern:

1. If there is a variable number of elements of the same type (or derived from) that share the same signature, handle them as a collection.
2. Show at the family architecture model the collection via a containment relationship with variable cardinality.
3. At the instance model subclass the component that contains the collection, and modify its cardinality via redefinition. Use subsetting to provide additional information if required.

Example

One of the parts in an airplane that can vary more in multiplicity is the wheels of the landing gear. A simple one, like the tailwheel-type, has three wheels, two at the front and one at the back. More complex configurations, like the one installed in the Antonov AN-225 Mriya [47], can have up to 32 of them. Furthermore, there are many types of wheels; some of them are suited for heavy duty, while others are a better fit for small loads. Figure 6.8a shows a BDD that models this example. The *Airframe* can have from 3 to 32 wheels, and each of them can be of *Remountable Wheel* or *Divided Wheel* types. Figure 6.8b, on the other hand, displays an instance whose airframe has three wheels, two at the front and one at the back.

Note that the blocks in this example do not contain behavior. If they did, they would need to have the same signature for this pattern to work.

Discussion

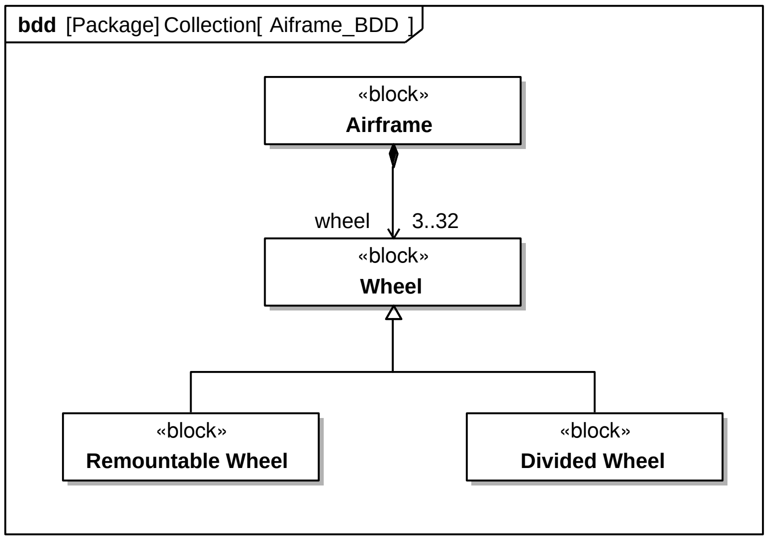
This pattern allows the model to manage a variable number of elements of the same type as a collection with a negligible impact on the rest of the system. The only possible side effect can happen if by subsetting an element the internal structure changes. When that happens, the new configuration needs to be modeled in the IBD of the instance.

Also, it is important to remember that for this approach to work it requires the collection cardinality to be at least one. When it can also be zero, it must be combined with the optional component pattern.

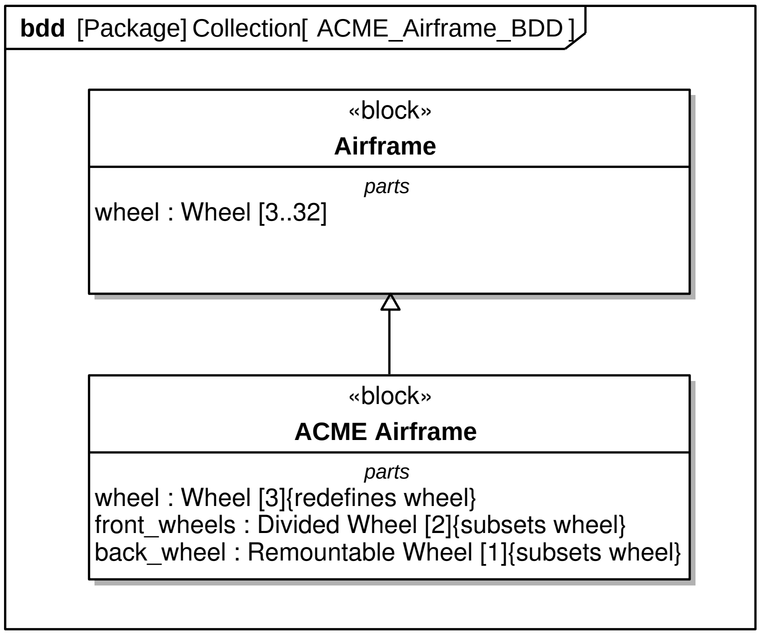
6.5 Data Flow Organization

6.5.1 Scenario Description

One of the biggest challenges faced by companies while developing a system is keeping data and control flows in check. A typical product will have hundreds if not thousands of them, each of which is likely to have different types, origins, destinations, and restrictions. Hence, managing these interfaces for one product is very tough work. This problem becomes exacerbated in a software product line, where each variant will have different combinations of flows.



(a) Airframe BDD


bdd [Package] Collection[ACME_Airframe_BDD]
«block»
Airframe
parts
wheel : Wheel [3..32]
«block»
ACME Airframe
parts
wheel : Wheel [3]{redefines wheel}
front_wheels : Divided Wheel [2]{subsets wheel}
back_wheel : Remountable Wheel [1]{subsets wheel}

(b) BDD of instance with collection.

Figure 6.8: Collection Pattern

Models can ease this problem by showing for each flow its origin, destination, type, and how they vary from product to product. However, for this representation to be helpful it needs to meet the following goals:

- The representation used must be as clean and clear as possible; when a diagram gets cluttered because of an unreasonable amount of connectors it loses its usefulness. Therefore, data flows need to be depicted in a concise way.
- It is important to reduce unnecessary manual work. In particular, changes in flows should not scatter throughout the model.
- It must be easy to identify the origin and the destination of the data. Furthermore, they must be kept synchronized.

6.5.2 Suggested Pattern

The first part of this pattern requires creating a data dictionary. The idea is to provide a central repository that describes the properties and data types used throughout the model. Each entry in the dictionary should contain the name of the property, its type, acceptable ranges, default values, description, origins and destinations. This information can become invaluable during the system design; it can help avoid duplicate variables, enforce consistent naming, and identify adequate values, among others. It is important to highlight that the actual dictionary does not necessarily have to be created manually and stored in one central location. If there is enough tool support, data can be placed at different locations in the model, and queried on demand to build the dictionary on the fly.

Next, the connectors in the model need to be put under control, which can be accomplished by using proxy ports typed with interface blocks. In them, this approach employs flow properties to describe atomic values and proxy ports for nested flows. The fundamental idea is to have only one port for all inputs of a block and one for all outputs. The input port contains flow properties for all the values directly required by the block, and one proxy port (nested interface blocks) for the input of each of its parts. Similarly, the output should contain flows properties for the values produced by the block and proxy ports for each of the parts.

Based on feature selections, some flows of data might not be required for all variants. A model can depict this optionality by setting the corresponding element of the interface block as [0..1]. If an atomic value is variable, its flow property is marked as optional. On the other hand, when a part of the block is not always required, the corresponding proxy port is marked as optional too.

The last part of the pattern deals with interface compatibility analysis, which involves making sure that the data required by a block and its part is indeed provided by the components it is connected to. The modeling tool should be able to perform this analysis automatically. To do so, it must decompose the interface blocks into atomic flow properties and match inputs by name, type, and direction. If there is a problem, the tool should report a compatibility violation. Note that the SysML specification does not provide detailed information about how to perform this analysis, so this could be an extension to the specification.

In summary, handling variability with the data flow organization pattern requires these following steps:

1. Create a central data dictionary.
2. Organize data and control signals in aggregate flows.
3. Use proxy ports and flow properties cardinalities to show data flow optionality.
4. The modeling tool should perform compatibility analysis by decomposing proxy ports (nested interface blocks) into atomic flow properties.

Example

Figures 6.9 and 6.10a show pieces of the *Controller Software* block in a BDD and IBD respectively. Even in such a small example the flow of data is very complex. The *Input Processing* block receives data, transforms it and produces clean flows for other components to use. The internal blocks, *State Mgr* and *Power Mgr*, use this data to perform calculations that can help control the system. Finally, the information that they produce is provided to the *Output Processing* block to transmit externally.

Without proper organization, keeping data flows under control would be very hard. For instance, the *Ice Protection Mgr* block is a Customization Point, meaning that each variant must define its particular implementation, potentially changing its signature in the process. In the worst case, any change to this interface would impact the containing blocks *Propeller Mgr*, *Power Mgr*, and the components they connect to; this would be very inefficient.

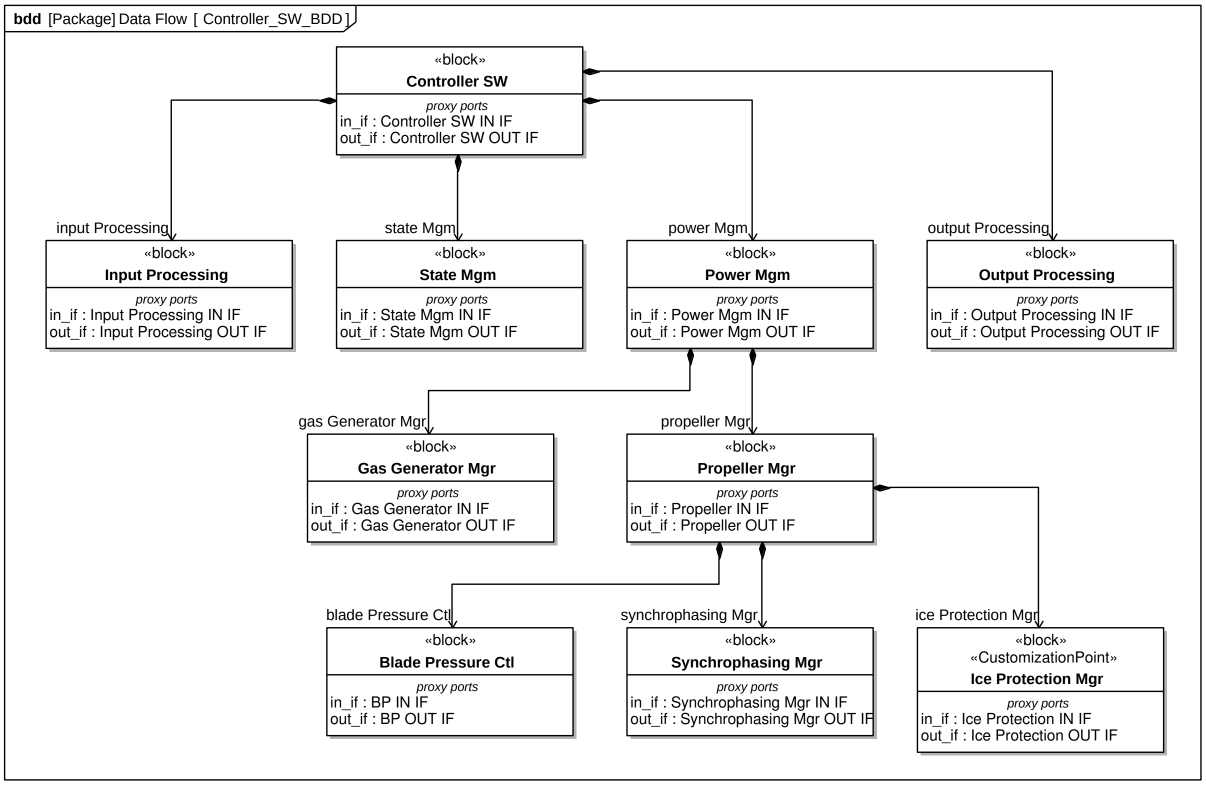
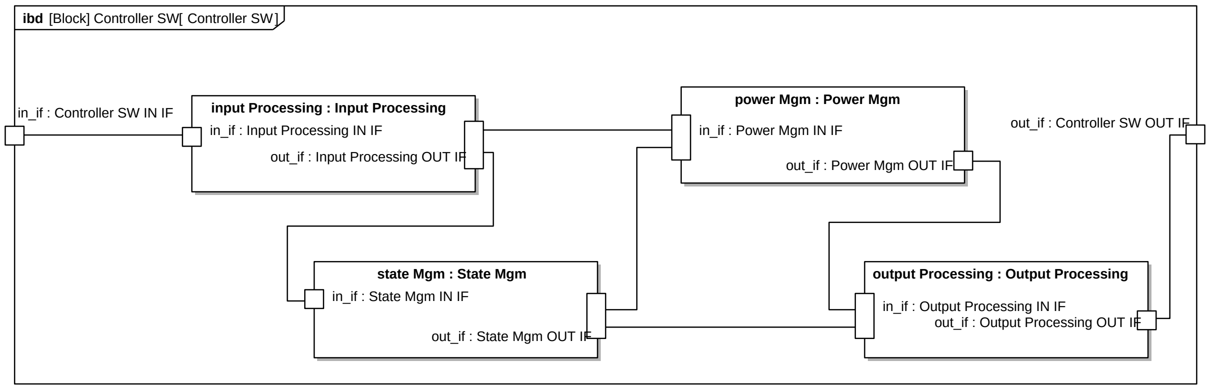
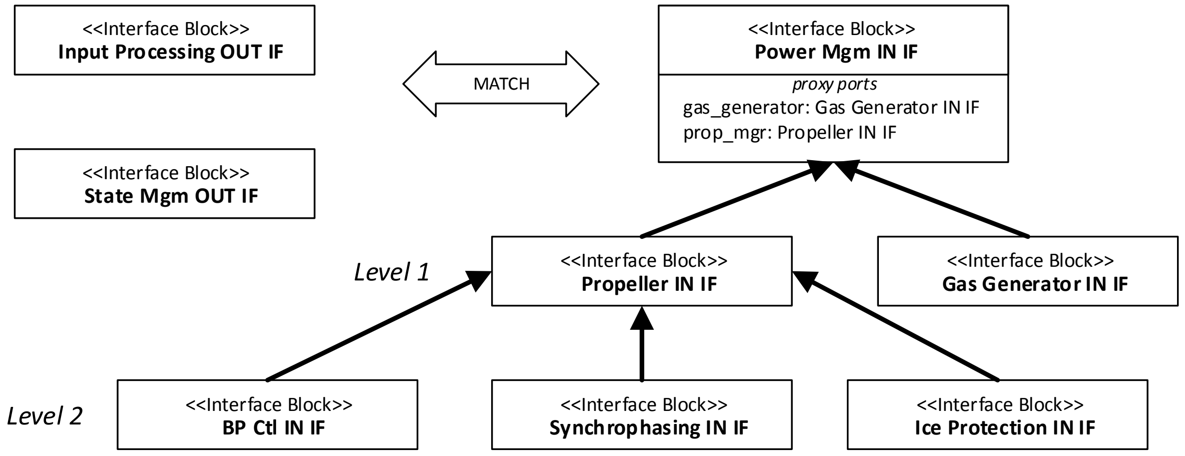


Figure 6.9: Controller SW BDD


ibd [Block] Controller SW[Controller SW]
in_if : Controller SW IN IF
input Processing : Input Processing
in_if : Input Processing IN IF
out_if : Input Processing OUT IF
power Mgm : Power Mgm
in_if : Power Mgm IN IF
out_if : Power Mgm OUT IF
out_if : Controller SW OUT IF
state Mgm : State Mgm
in_if : State Mgm IN IF
out_if : State Mgm OUT IF
output Processing : Output Processing
in_if : Output Processing IN IF
out_if : Output Processing OUT IF

(a) Controller SW IBD



(b) Interface Compatibility Analysis

Figure 6.10: Data Flow Organization Pattern

This problem can be solved by using the data flow organization pattern. Data must be organized in aggregate flows as shown in Figure 6.10b. With this structuring, each component does not need to worry about data origin or destination; it needs only to state what information it needs to operate and which one it will produce as output. Later, during interface compatibility analysis, the tool must retrieve the list of applicable flows properties. In this case, the list for *Power Mgm IN IF* would include all the ones it defines directly and the ones contained at level 1 and level 2. Once the list is ready, the tool only has to make sure that the inputs, *Input Processing OUT IF* and *State Mgm OUT IF*, provide all of these flow properties.

Discussion

Using this pattern provides the designers with better data organization. It allows identifying duplicate variables, enforcing system-wide policies, like naming conventions, and keeping types, ranges, and values consistent. Moreover, organizing data in nested interface blocks allows isolating most of the model from updates to data flows. Whenever a flow is added, removed, or modified, the change will affect only its producer and consumer. Without this approach, all intermediate connections would need to be updated to accommodate the modification.

Finally, this pattern can be particularly beneficial if the model is used to generate skeleton code. Nested interface blocks can be decomposed and used to wire components automatically, which can significantly reduce manual work and could potentially prevent errors.

6.6 Bus Alternatives

6.6.1 Scenario Description

Except for the most trivial cases, systems do not work in isolation. They cooperate with other entities, like external systems, users (actors), and even with their surrounding environment. To make this interaction possible, the communication between them must take place in a medium common to the sender and receiver, and requires following well understood rules. Unfortunately, technical constraints, equipment choices and other factors, can modify the way they interact. Worse still, communication is so ubiquitous that even the smallest changes can impact multiple parts of the system. Modeling has the potential to help reduce this problem by aiding in the creation of a design resilient to these changes. Such a representation must meet the following goals:

- Design should support changes in communication technologies, protocols, and equipment.
- Reduce the amount of manual work as much as possible. Ideally, changes should span only one well-defined layer.

6.6.2 Suggested Pattern

This pattern relies on several advanced constructs of SysML present in BDD and IBD diagrams. Namely, association blocks, proxy, full, and nested ports; refer to appendix A for more information about each of them.

The key idea is to use full ports as a boundary between the internal system and the external entities; they will deal with the intricacies of transforming protocol specific messages into usable data and vice-versa. This way the rest of the system will deal only with clean data, and all changes pertaining communication will be constrained to full ports.

The first step consists of specifying the internal structure of each of the full ports. Since blocks are their only acceptable type, two of them are created; one for each endpoint. Then, each block gets two proxy ports: one used to provide usable data and another one for protocol specific messages. Finally, the internal structure is described in an IBD as shown in figure 6.12. In this example, it is assumed that only two components are required to accomplish this conversion, but more complex structures are acceptable; what is important is that these elements remain within the port.

Next, it is important to define how to link the two endpoints. Probably the best way to do so is via association blocks. This construct specifies that any two participants connected with it must communicate as specified in the corresponding IBD. In this case data in one system gets transformed into a protocol specific message, transferred via the medium, and transformed back by the other endpoint.

Finally, the last step is to use these elements in the model. For example, figure 6.13 shows an engine communicating with the aircraft airframe via an avionics bus. Each endpoint is a full port defined by one of the blocks previously mentioned, and the association is typed as an *Avionics Bus*.

At the instance model, the block that holds the element whose communication is subject to modification is subclassed, and the port redefined as appropriate. These are the possible variations:

Bus Change: The communication protocol or equipment might change. For instance, in avionics the communication bus can be ARINC 429 [2] or MIL-STD-1553 [24]. This is the most disruptive change as it requires updating elements 1, 2, 3, 4, 5, 6, and 7 as shown in figures 6.11 and 6.12.

Data Differences: The data being transferred might change because of other types of variability. Elements 5 and 7 would need to be modified to address it.

Recipient Differences: External changes can affect the structure of the data sent, even if the protocol is the same. For instance, ARINC 429 requires sending data organized by labels; an engine might need to use different labeling strategies depending on the aircraft it is mounted on.

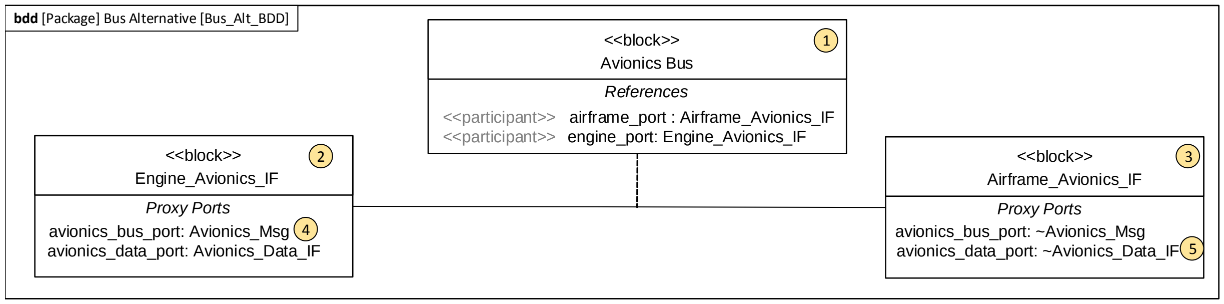
In summary, these are the steps required to implement this pattern:

1. Use proxy ports as system boundaries. Describe their internal structure as blocks.
2. Show how to connect ports with association blocks.
3. Use elements in IBD to qualify communication.
4. Modify as necessary via subclassing and redefinition.

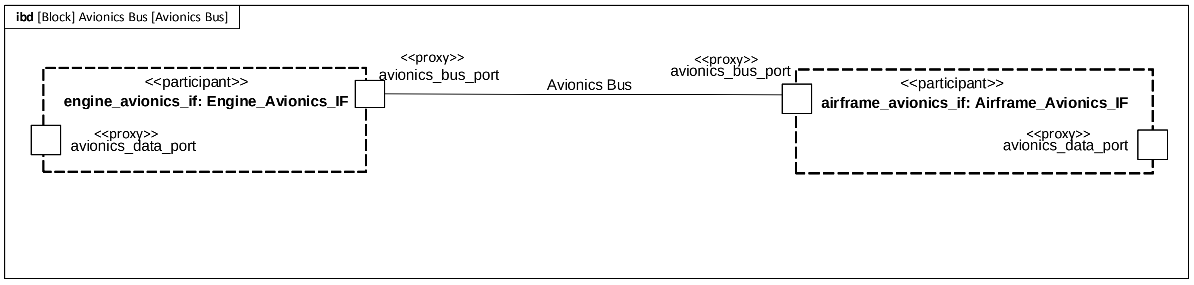
Example

Figure 6.13 shows how the engine can be connected to the airframe via an avionics bus. As mentioned previously, there are more than one possible communication standards for this interaction, like *ARINC 429*, *MIL-STD-1553*, among others. Therefore, it makes sense to isolate this part from the rest of the model.

Each possible communication standard will be different, but they still share some things that can be leveraged while modeling. First, there will be an endpoint on the airframe side and another on the engine, which will be linked by some connector. This information is depicted via the association block *Avionics Bus*, representing the bus per se, and *Engine_Avionics_IF* and *Airframe_Avionics_IF* which serve as the endpoints (called participants in SysML). The second thing in common is that the communication between the endpoints will be carried through standard-specific messages; the model portrays this with the *Avionics_Msg* data type. Third, the system on one side must transform the data to send from a protocol-independent structure to the *Avionics_Msg* type, transmit it, and

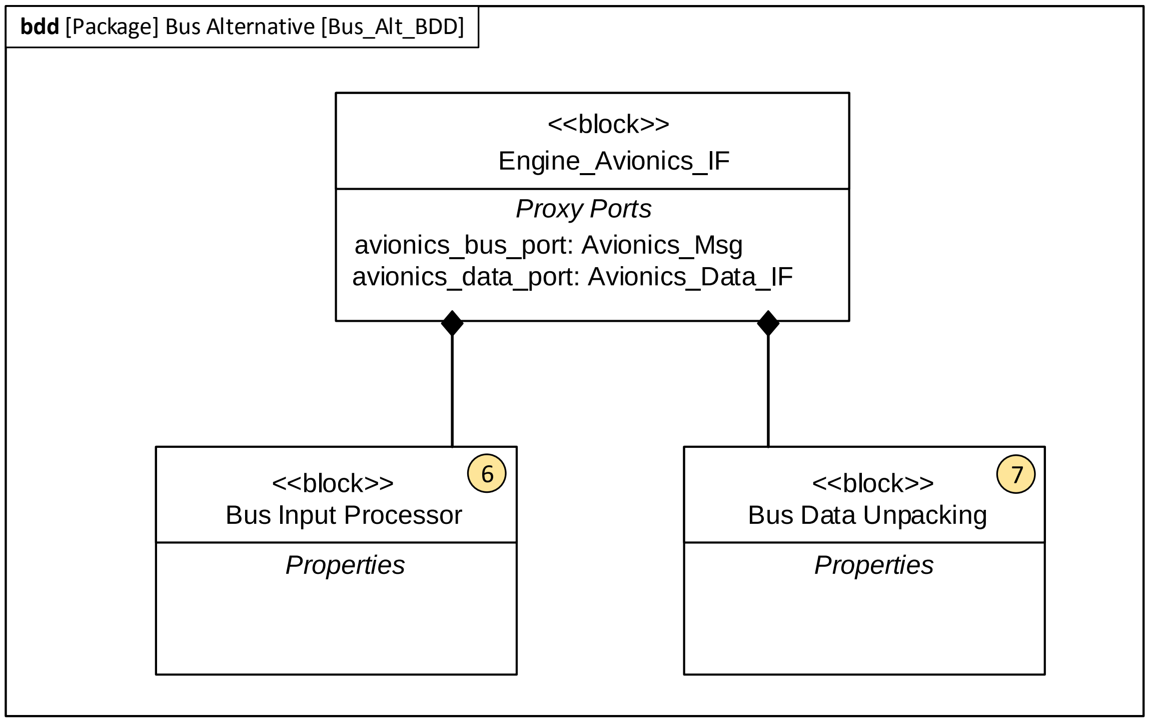

bdd [Package] Bus Alternative [Bus_Alt_BDD]
<<block>>
Avionics Bus
1
References
<<participant>> airframe_port : Airframe_Avionics_IF
<<participant>> engine_port: Engine_Avionics_IF
<<block>>
Engine_Avionics_IF
2
Proxy Ports
avionics_bus_port: Avionics_Msg
4
avionics_data_port: Avionics_Data_IF
<<block>>
Airframe_Avionics_IF
3
Proxy Ports
avionics_bus_port: ~Avionics_Msg
avionics_data_port: ~Avionics_Data_IF
5

(a) Association Block BDD

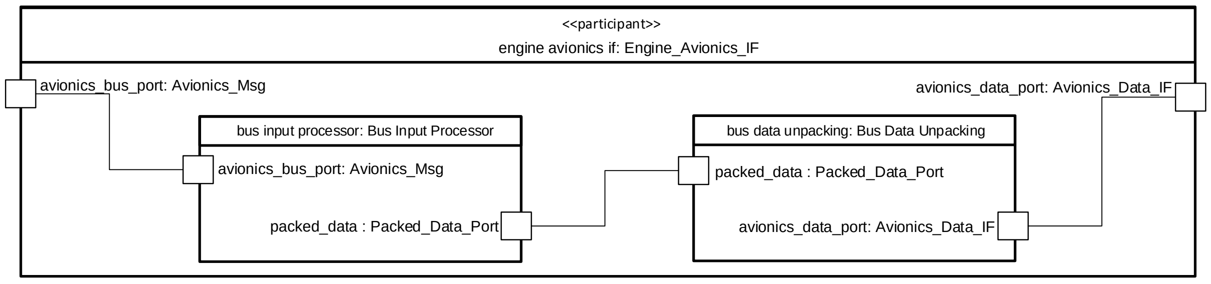


(b) Association Block IBD

Figure 6.11: Bus Alternatives - Association Block



(a) Endpoint BDD


<<participant>>
engine avionics if: Engine_Avionics_IF
avionics_bus_port: Avionics_Msg
avionics_data_port: Avionics_Data_IF
bus input processor: Bus Input Processor
avionics_bus_port: Avionics_Msg
packed_data : Packed_Data_Port
bus data unpacking: Bus Data Unpacking
packed_data : Packed_Data_Port
avionics_data_port: Avionics_Data_IF

(b) Endpoint IBD

Figure 6.12: Bus Alternatives - Endpoint

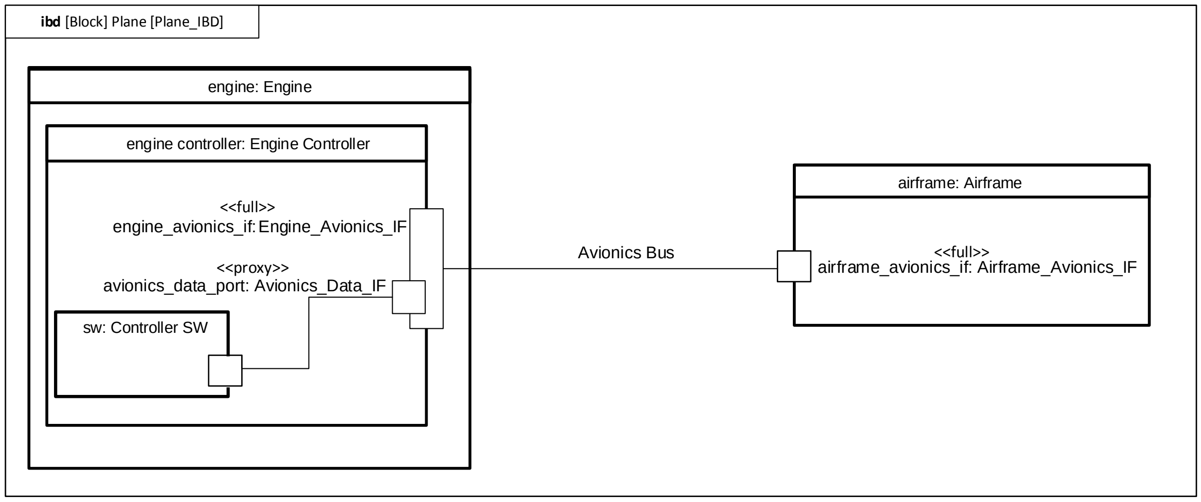

ibd [Block] Plane [Plane_IBD]
engine: Engine
engine controller: Engine Controller
<<full>>
engine_avionics_if: Engine_Avionics_IF
<<proxy>>
avionics_data_port: Avionics_Data_IF
sw: Controller SW
Avionics Bus
airframe: Airframe
<<full>>
airframe_avionics_if: Airframe_Avionics_IF

Figure 6.13: Bus Alternative Pattern

the other side convert it back to usable data; figure 6.12 depicts the transformation process in one endpoint, and figure 6.11 the communication. Finally, both endpoints will expose *avionics_ data_ port* to the rest of the system; this proxy port will provide data in a format ready to use.

Once this part of the model is in place, a particular bus, like ARINC 429, can be created just by subclassing *Avionics Bus*. At this point, any instance can make use of the new block and, if necessary, alter its structure by using redefinition. The possible changes are described in the previous section 6.6.2.

Discussion

Using this pattern allows the model to isolate the system from any communication related variability, provides guidance about what parts to modify depending on the variation at hand, and concentrates all changes in a well-defined layer. Since the interaction with external entities can vary because of a myriad of reasons, structuring the system in this way can allow it to become much more stable.

However, the disadvantage of this approach is that it requires a considerable amount of work to set up the initial structure. Furthermore, it requires some advanced SysML concepts, like association blocks, proxy, and full ports, some of which not all tools currently support.

6.7 Equipment Differences

6.7.1 Scenario Description

A system contains variability caused not only because of differences in functionality, but also because of the alternative equipment that can be used to accomplish it. For example, part of the operation of the engine controller consists in determining the right amount of fuel to use at any given time. A metering unit receives this value and moves mechanical parts to provide the requested fuel flow. However, the exact meaning of this value depends on the hardware used; in a stepper motor it represents the rotation of a disk; in a metering valve it denotes the position of a solenoid; other types of equipment can have alternative meanings too. Moreover, the value might also be dependent on the manufacturer. To be able to control the system properly, software must be aware of these differences and react accordingly.

Since external data is so prone to change, the system should be structured and modeled differently to mitigate this problem. Such representation should have the following characteristics:

- Equipment variation should not scatter throughout the system. The scope of any such change should be as small as possible.
- The model should provide guidance about what parts to modify depending on the type of change.

6.7.2 Suggested Pattern

Equipment differences can hardly be prevented; they are dependent on decisions beyond software control. However, systems can be protected to some degree by isolating them from any external changes. The idea is to make the internal components work with a well-defined set of inputs and outputs, and create two specialized layers that will perform the chore of adapting them to the actual hardware. The first layer will handle inputs; it will transform the values from device-specific units into a standard representation, will perform filtering, and some level of fault detection and accommodation. Similarly, the second one will focus on outputs; it will convert internal data into a format that the hardware can understand.

To make this work, first it is necessary to define the standard interfaces. As described in the data flow organization pattern, a central data dictionary can be used for this purpose.

For each input and output it must contain the name, description, valid range and type. This structure can be used to promote consistency throughout the system.

Next, it is necessary to define the input and output layers. Each one can be described as a SysML block, with proxy ports to connect them to the rest of the system. Both, the interface block of the ports facing outwards and the internal structure of these layers, will vary continually to respond to external equipment changes; this is acceptable, since the variation will be constrained to these parts. On the other hand, the ports facing to the insides of the system will be much more stable. Internal components can operate without worrying about data representation differences as long as they never access external flows directly, and instead make use of the flow properties defined in these ports.

In summary, these are the steps required to make use of this pattern:

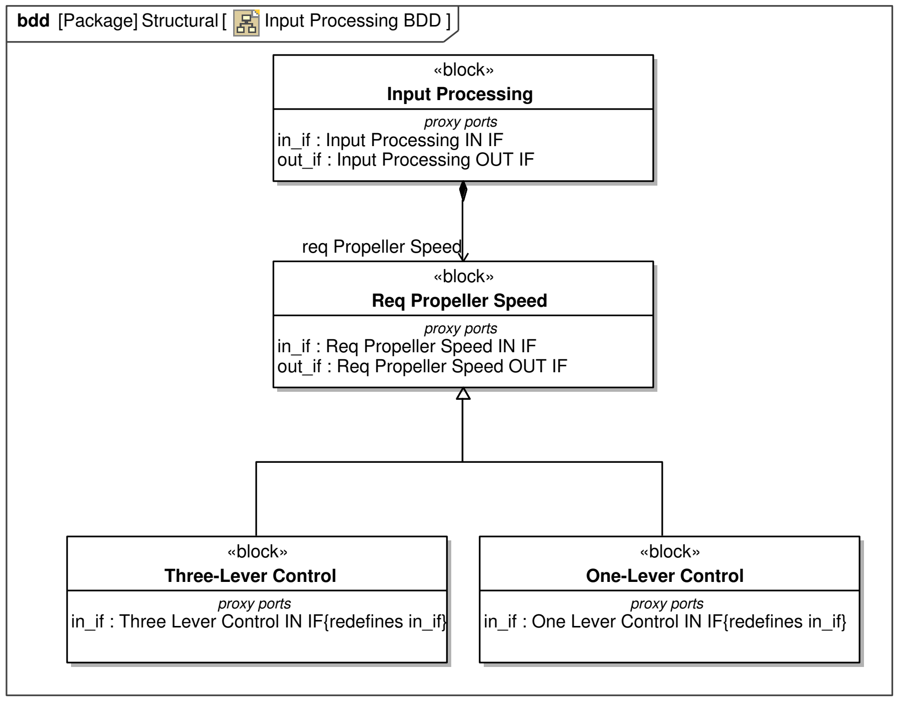
1. Define data dictionary to enforce consistency throughout the system.
2. Create input layer. It will deal with value conversion, filtering, and some fault detection and accommodation.
3. Create output layer. It will transform internal data into a format that the equipment can understand.

Example

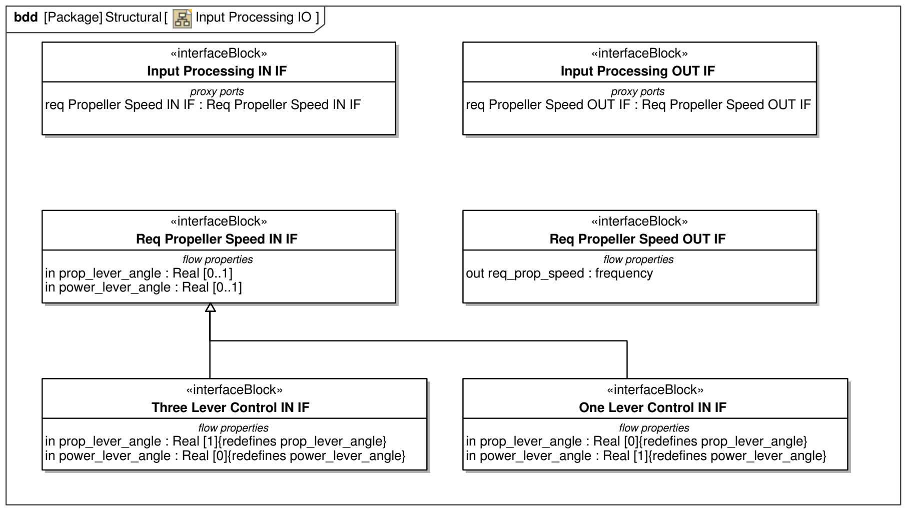
One type of variability a turboprop engine can present is the amount of control a pilot has over the propeller. In a very simplistic case, the cockpit has three levers, one of which determines the propeller RPM(revolutions per minute). However, there are some engine variants that can replace these levers with only one in an attempt to reduce the pilot's workload. The controller of such an engine has to determine the propeller rotation rate indirectly from the requested power.

The problem with this kind of variability is that the controller has to change the way it governs the propeller because of external changes, even though the internal operation should still be the same: adjust the blade angle to maintain a desired speed. Therefore, it is convenient to create a layer that will deal with the different lever configurations, and always provide the requested RPM in the same way. This approach allows the internal system structure to stay the same.

As displayed in the figure 6.14, the *Input Processing* layer has a block that deals with the conversion of the propeller speed. Note that since this block contains variability, the


bdd [Package] Structural [Input Processing BDD]
«block»
Input Processing
proxy ports
in_if : Input Processing IN IF
out_if : Input Processing OUT IF
req Propeller Speed
«block»
Req Propeller Speed
proxy ports
in_if : Req Propeller Speed IN IF
out_if : Req Propeller Speed OUT IF
«block»
Three-Lever Control
proxy ports
in_if : Three Lever Control IN IF{redefines in_if}
«block»
One-Lever Control
proxy ports
in_if : One Lever Control IN IF{redefines in_if}

(a) Input Layer BDD


bdd [Package] Structural [Input Processing IO]
«interfaceBlock»
Input Processing IN IF
proxy ports
req Propeller Speed IN IF : Req Propeller Speed IN IF
«interfaceBlock»
Input Processing OUT IF
proxy ports
req Propeller Speed OUT IF : Req Propeller Speed OUT IF
«interfaceBlock»
Req Propeller Speed IN IF
flow properties
in prop_lever_angle : Real [0..1]
in power_lever_angle : Real [0..1]
«interfaceBlock»
Req Propeller Speed OUT IF
flow properties
out req_prop_speed : frequency
«interfaceBlock»
Three Lever Control IN IF
flow properties
in prop_lever_angle : Real [1]{redefines prop_lever_angle}
in power_lever_angle : Real [0]{redefines power_lever_angle}
«interfaceBlock»
One Lever Control IN IF
flow properties
in prop_lever_angle : Real [0]{redefines prop_lever_angle}
in power_lever_angle : Real [1]{redefines power_lever_angle}

(b) Input Layer Interfaces

Figure 6.14: Equipment Differences Pattern - Family Architecture Level

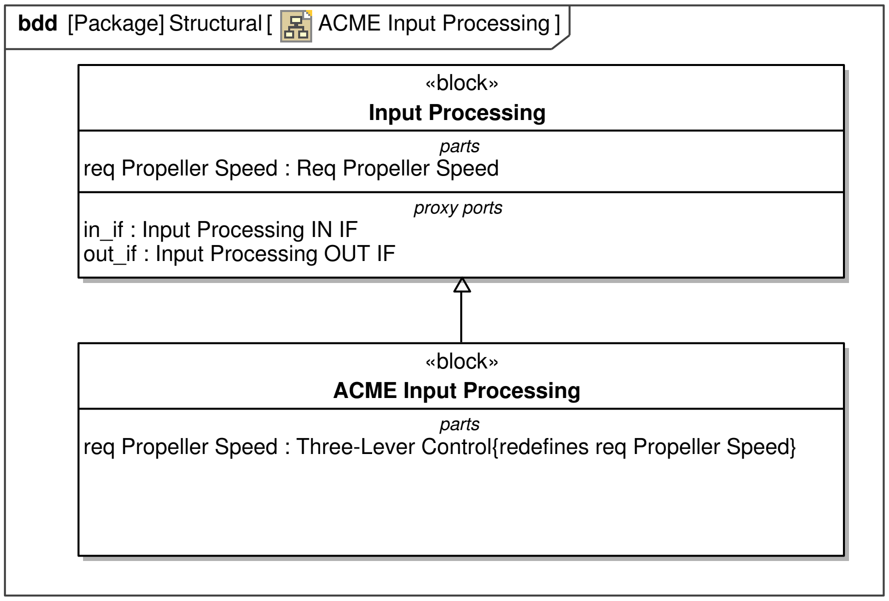

bdd [Package] Structural [ACME Input Processing]
«block»
Input Processing
parts
req Propeller Speed : Req Propeller Speed
proxy ports
in_if : Input Processing IN IF
out_if : Input Processing OUT IF
«block»
ACME Input Processing
parts
req Propeller Speed : Three-Lever Control{redefines req Propeller Speed}

Figure 6.15: Equipment Differences Pattern - Instance Level

alternative component pattern is used, and two subclasses are created; one deals with the three-lever configuration and the other with the one-lever case. Internally, each of them will transform the input into the expected value, will perform filtering, and some levels of fault-detection and accommodation. Finally, figure 6.15 shows the input layer of the ACME instance, which will use the *Three-Lever Control*; the rest of the system is unaware of which configuration is employed, since the provided *req_prop_speed* value is always the same.

Discussion

A model that makes use of this pattern can limit the impact that the exterior has over the structure of the system; they can almost evolve independently. Whenever such a change happens, its impact will be scoped to the input/output layers, leaving the rest of the system unaffected. However, this approach might make some internal adjustments a little bit harder to perform. Changes necessary for adding or removing internal components might not only encompass the component itself anymore, but also the input/output layers. Moreover, modifying the data crossing these layers requires careful handling, since any update can affect more than one part. This last problem can be mitigated to some degree by keeping a data dictionary; it would allow to know which components would be affected by changing a data item.

6.8 Summary

This chapter described how variability can occur in structural diagrams, such as block definition and internal block diagrams, and proposed patterns to deal with it. First, it introduced the variable property. It makes use of value properties for quantitative variability, subclassing for intervals and PDIs for data that will be provided in the future. Second, it provided two patterns for optional elements: the component removal and the mock component patterns. Next, it provided two ways to deal with alternative behavior. The XOR pattern is suggested for situations where one component can have only one of the alternative implementations. The collection, on the other hand, is useful when instances can have one or more components selected at the same time. Similarly, the chapter proposed the data flow organization pattern to handle interfaces differences throughout the model. Also, it provided the bus alternatives approach to manage communication changes. Finally, it presented the equipment differences pattern to deal with external changes.

Chapter 7

SCADE Patterns

This chapter takes the focus away from the models and instead centers around the methodologies and mechanisms necessary to make variability handling at the implementation side compatible with the modeling patterns previously described. The key idea is to allow the creation of code that can be later customized to each of the supported variants. The original version must reflect the family architecture model and the customizations, on the other hand, should follow the changes described in the instance models.

SCADE Suite [42] is a model-based development environment that can help streamline the certification process. Since this tool can reduce costs significantly, it is becoming one of the implementation languages of choice in the avionics domain. Since the focus of this work is also around avionics, the patterns presented in this chapter are given in this language. However, each one of them also contains a description of what the problem is, and what constructs are necessary to solve them; hopefully this approach will make the information presented more generic.

7.1 Alternative Nodes — Same Signature

7.1.1 Scenario Description

Some of the modeling strategies, such as the mock component pattern, require swapping parts of the system that retain the same interfaces, but that provide alternative functionality. In object-oriented programming languages, this situation can be easily solved via inheritance and polymorphism. However, not all implementation languages have these advanced features. Therefore, to allow this type of variability the tool used should provide the following features:

- Allow encapsulation of alternative behavior in separate components.
- It should be clear that the alternatives share the same interface.
- The code should be able to change between implementations without having to modify the calling component.

7.1.2 Suggested Pattern

Even though SCADE does provide some mechanisms for genericity and polymorphism, they behave differently from how they usually do in object-oriented languages, so a different approach must be devised to solve this problem.

One possible solution involves making the replacement of components by hand. The idea is to create one project that contains the part of the code that will not change, and refer to the variable behavior as a call to operators in external libraries. In particular, each alternative needs to be stored in a separate project; the appropriate one is selected depending on the functionality to provide. For this approach to work, the operator location, name and signature must be the same. Otherwise, SCADE is unable to locate the items to swap.

It is important to note that although this replacement can be performed manually, the process can certainly be automated. Current versions of SCADE store the implementation information, like operators and packages, in XML-based files. Therefore, it is possible to use scripts to modify the XML element that contains the location of the libraries. If these scripts are connected to the feature model, the whole configuration process can become fully automated.

In summary, these are the steps required to provide alternative implementations with the same interface in SCADE:

1. Place alternative behavior in operators stored in separate SCADE libraries. Each of them shall have the same path, name, and interface.
2. The common code shall make a call to the variable operator. Since the interface is kept unchanged, this part of the implementation can remain oblivious of the actual library to use.

3. Swap the component depending on the desired functionality to provide; this task can be performed manually or through a script.

Example

Figure 7.1 contains the implementation of the model described in figure 6.4a. As shown, depending on the status of the synchrophasing feature the propeller operation can be modified by two alternative behaviors. The first one, encapsulated in library *SynchrophasingConcrete*, synchronizes the propeller phase to reduce noise and vibration. The second one, included in *SynchrophasingMock*, keeps the propeller unchanged.

PropellerMgr makes the call without caring about which library is in place; SCADE will take care of the swap as long as the operators in both libraries keep the same location, name and signature.

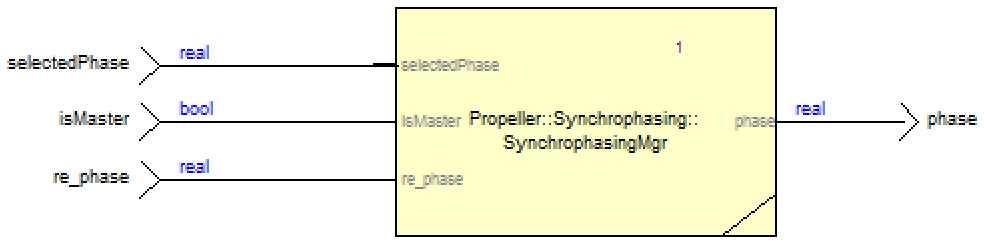
Discussion

This pattern allows the creation of a customizable component that can choose from alternative behavior as long as each of the options keeps the interface unchanged. For this approach to work, the different implementations must be stored in independent libraries, and rely on external mechanisms, like custom scripts, to make the necessary file replacements. At least in the avionics domain, this flexibility does not come for free; the configuration process might need to be verified to ensure that the right selection is being performed, and potentially, will need to be certified.

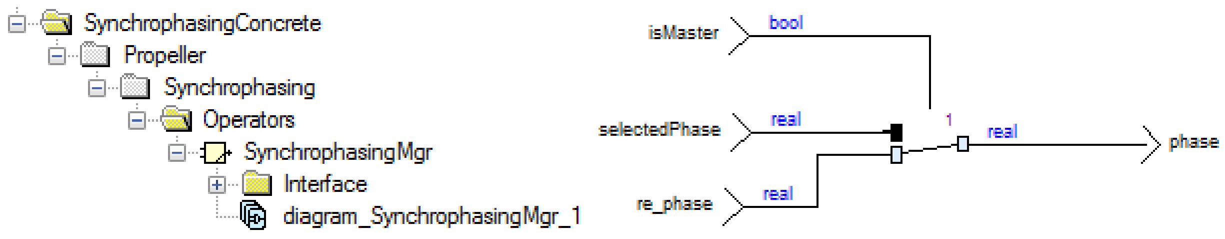
7.2 Alternative Nodes — Different Signature

7.2.1 Scenario Description

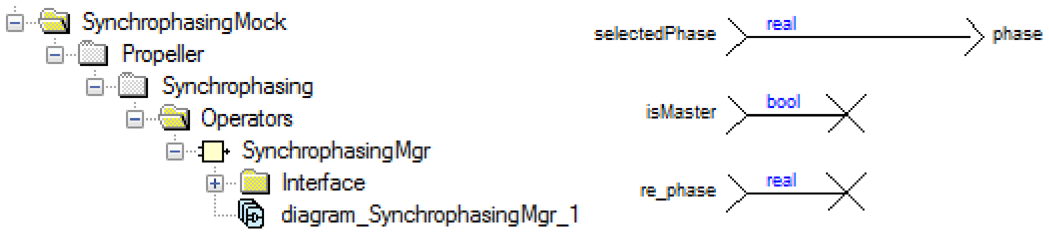
As mentioned in the previous pattern, it is relatively simple to provide alternative behavior when they share the same interface; only swapping the components is necessary. However, if the interface requires modification then the process becomes harder. The biggest problem is that any changes to the inputs or outputs need to be reflected in the calling operator. Even worse, it might be necessary to carry over the modifications even beyond the parent operator; this can get out of control easily.



(a) Calling Operator



(b) Alternative Operator 1



(c) Alternative Operator 2

Figure 7.1: Alternative Node - Same Signature

Therefore, to manage variability there shall be a way of dealing with interface changes efficiently. Such an approach should have the following characteristics:

- It shall be possible to swap components even if they have different interfaces.
- Changes to the functionality and to the interfaces must be kept in synch.
- There should not be any need to modify manually any of the parent components, including the calling operator.

7.2.2 Suggested Pattern 1

The pattern described in this section has not been implemented, as it requires a construct that is currently unavailable in SCADE. However, it might be helpful to consider implementing it as it could make this type of variability remarkably efficient.

The cornerstone of this approach is the swapping of components, described in section 7.1, and an abstract struct. The idea is still to make use of alternative libraries that will be interchanged as necessary. All the constant inputs/outputs will behave as normal, but the ones that can vary will flow within the aforementioned abstract struct.

Each library will offer different functionality, and will have a unique implementation of the abstract struct based on the data that it needs; it can be empty if no data is necessary. Furthermore, the abstract struct shall allow nesting. This way interface variability can happen at different levels, but each component only needs to be aware of the changes at its level.

After a particular library is chosen based on the functionality to offer, the KCG generator can then produce code as normal. The only extra effort is to decompose the abstract struct into atomic elements, and wire them based on name and type. This process is shown in figure 7.2b.

Putting everything together, these are the envisioned steps to make this approach work:

1. Place alternative behavior in separate libraries. Each library shall define the variable data it needs in an abstract struct.
2. During code generation, decompose abstract struct and perform auto-wiring based on name and type.
3. Keep data producers and consumers in synch.

Example

Figure 7.2 shows how the proposed interface compatibility analysis would work between *Operator_1* and *Operator_a*. Figure 7.2a focuses on explaining how SCADE nodes would be connected. On the left side, *Operator_1* produces an output obtained by calling *Operator_2* and *Operator_3*, which make other calls themselves. *Operator_a*, on the other hand, will receive that as input and will distribute it to its child nodes, *Operator_b* and *Operator_c*, and to the grandchild node *Operator_d*. Figure 7.3, describes the way the abstract structs are nested to allow this design.

Finally, figure 7.2b presents how items are matched. Each nested struct is recursively decomposed until only atomic properties are left. At that point, the output variables are associated with the input ones by comparing their names and types. If no errors are found, SCADE should wire them automatically.

7.2.3 Suggested Pattern 2

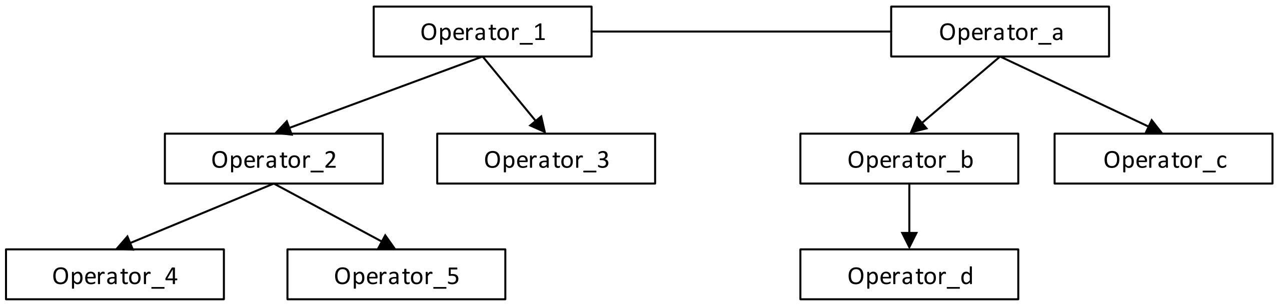
The approach described in this section is inspired by the adapter pattern [22, p. 139] used in object-oriented programming languages. The idea is that instead of adjusting the calling operator based on the interface of each alternative, an intermediate adapter will handle the differences.

As explained in section 7.1, each possible implementation will be stored in different libraries. However, for this pattern each one will also include the adapter necessary to transform the external interface into the one needed internally. Since the adapters are the ones responsible for calling the internal operator, they need to share the same name, inputs/outputs, and location so that SCADE can identify them. The operators, on the other hand, are free to change.

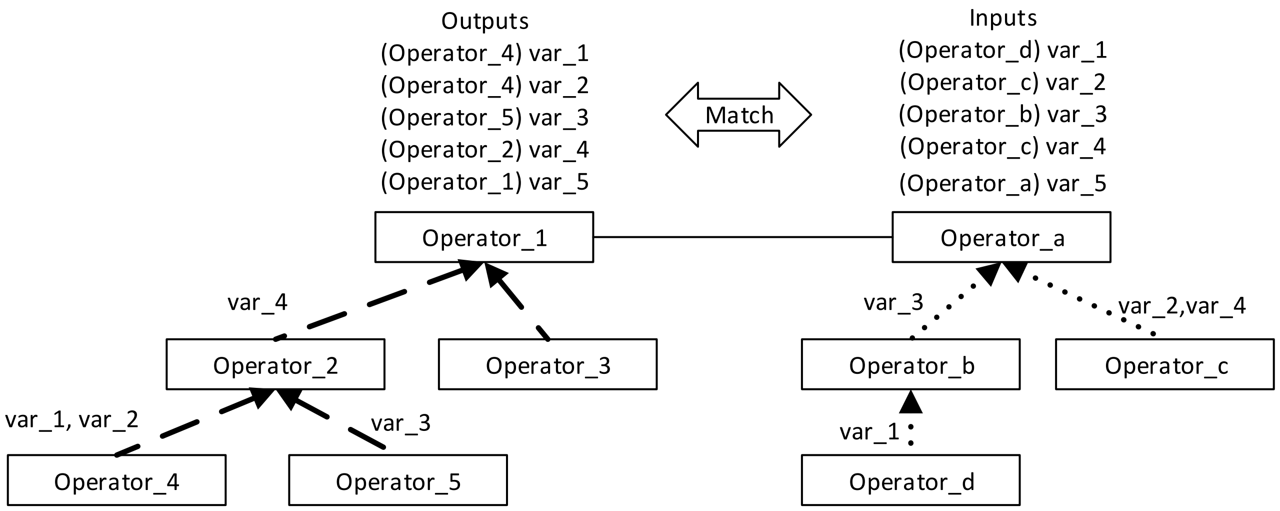
Most of the times the role of the adapter will only be terminating unused inputs, but more sophisticated approaches are possible. Note that this redirection might have a small, but still noticeable performance impact. If deemed necessary, the operator itself can handle the interface adaptation; a design choice has to be taken to either favor design cleanliness or performance.

These are the steps required to use this pattern:

1. Place alternative implementation in different libraries.



(a) Operator Calls



(b) Input/Output Matching

Figure 7.2: Interface Compatibility Analysis

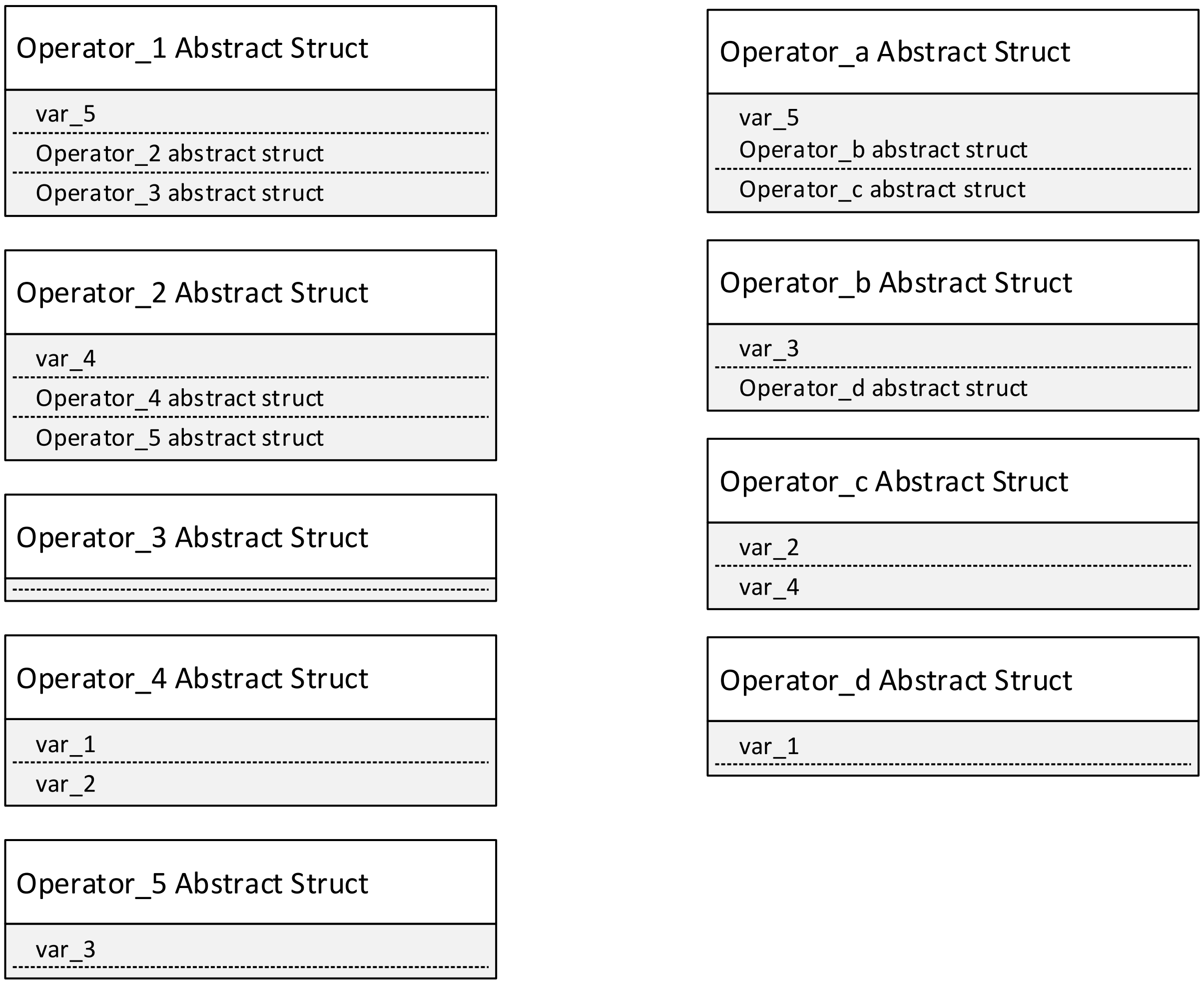

Operator_1 Abstract Struct
var_5
Operator_2 abstract struct
Operator_3 abstract struct
Operator_2 Abstract Struct
var_4
Operator_4 abstract struct
Operator_5 abstract struct
Operator_3 Abstract Struct
Operator_4 Abstract Struct
var_1
var_2
Operator_5 Abstract Struct
var_3
Operator_a Abstract Struct
var_5
Operator_b abstract struct
Operator_c abstract struct
Operator_b Abstract Struct
var_3
Operator_d abstract struct
Operator_c Abstract Struct
var_2
var_4
Operator_d Abstract Struct
var_1

Figure 7.3: Abstract Struct Nesting

2. Create an adapter for each library. Each one of them must have the same name, interface, and location, and will be in charge of transforming the provided interface into the one needed internally.
3. Change the desired functionality by swapping libraries; this task can be performed manually or through a script.

Example

The example in figure 7.4 shows how the propeller controller calls the *BladePressureCtl* operator to get the pressure to use at any time. There are two possible implementations depending on the type of feedback available in each engine variant. If the current blade angle is available, *BPClosedLoop* library is used, and the *propSpeed* argument is ignored. On the other hand, if this piece of data is not accessible then the calculation is performed with the *BPOpenLoop* library, which makes use of *propSpeed*, but that disregards *currentBeta*. The calling controller is ignorant about the fact that each operator requires a different interface; the adapters are the ones that deal with this issue.

Discussion

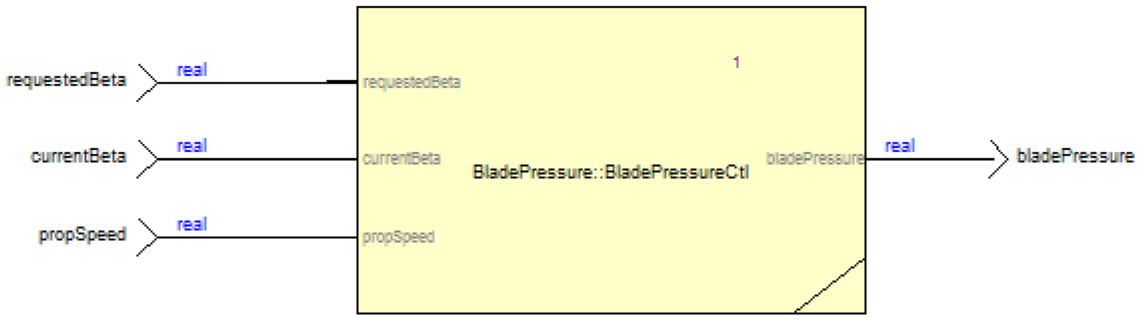
This approach allows the creation of customizable components with parts of behavior that can change between a set of possible alternatives, even if their interfaces change. Probably, the biggest advantage is that the calling operator does not require any change; the functionality can be swapped simply by importing a different library.

However, this flexibility comes at a price. First of all, adapters need to be created for each alternative to support. Second, if one of the alternatives requires changing its input/output, the adapter of the other ones must receive the update too, even if they do not need that data at all. Finally, variants might end up with pieces of useless data flowing through them, which could affect performance in extreme cases.

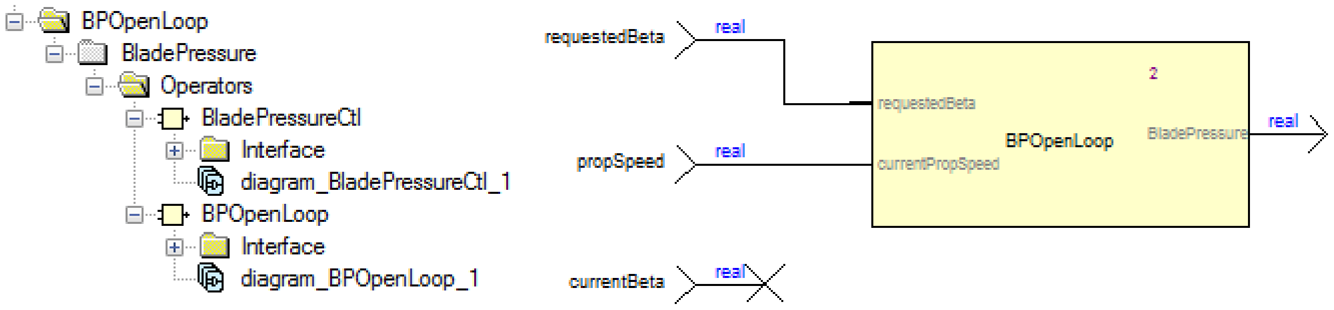
7.3 Optional Node

7.3.1 Scenario Description

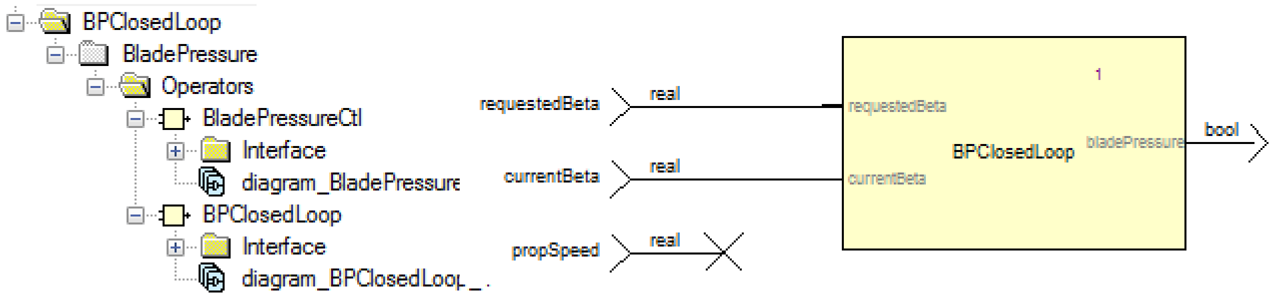
One common type of variability happens when a part of the system gets added or removed altogether based on the feature selection. At the model level, this case is handled with the



(a) Calling Operator



(b) Alternative Operator 1



(c) Alternative Operator 2

Figure 7.4: Alternative Node - Different Signature

optional component pattern, described in section 6.2; such an approach has to be reflected on the implementation side too. This task can get complicated if enabling the component can trigger changes in the calling operator.

An efficient method to handle this scenario should have the following characteristics:

- It should allow adding or removing optional components as necessary.
- It should have little or no impact on the calling operator.
- Manual work shall be minimized as much as possible.

7.3.2 Suggested Pattern

This scenario is very similar to the one generated by swapping components with varying interfaces. In fact, it could also benefit from using the abstract struct as envisioned in section 7.2.2. The operator that includes the optional component could organize all the inputs/outputs of the optional component in this struct. If the option were enabled, the struct would be decomposed into atomic flows and wired automatically. If it were disabled, then the struct would be empty, and no elements would require connections. Unfortunately, SCADE does not currently provide this kind of construct, so a different approach is necessary.

The biggest challenge to minimize the manual work is making sure that the calling operator will not require modifications because of the status of the optional component; it should always call the operator the same way regardless if the component is on or off. One way to do this is following the same strategy as section 6.2.2: using a mock.

The idea consists in having two alternatives with the same interface, placed on different libraries, for each optional component. One of them will have the real functionality while the other one will only return default values. If the optional feature is enabled, the library with the concrete implementation must be imported. Otherwise, the mock library is the one to use.

In summary, these are the measures to take to use this pattern:

1. Create two libraries, one with the real implementation, and another with a mock node. Both of them must have the same interface.
2. Select which library to import based on the feature selection. If the option is enabled select the real implementation. Otherwise, use the mock node.

Example

The example included in figure 7.5 describes the component *PhaseValidator*, which ensures that the measured phase angle is within a valid range. However, this functionality is required only if the engine contains a sensor to make this measurement. Otherwise, the data contained in this flow is useless and must be ignored. When the sensor is present, the library that contains the real implementation, displayed in figure 7.5b, must be selected. On the other hand, if it is not present, then the mock is sufficient; this is shown in figure 7.5c.

Discussion

This pattern allows enabling and disabling an optional component as necessary without disturbing the calling operator. Unfortunately, it does not adapt the interfaces of the components, so unnecessary data might be kept flowing throughout the system.

Note that to make both, the model and the implementation consistent, if the mock pattern is used during modeling, that same approach should be adopted in SCADE.

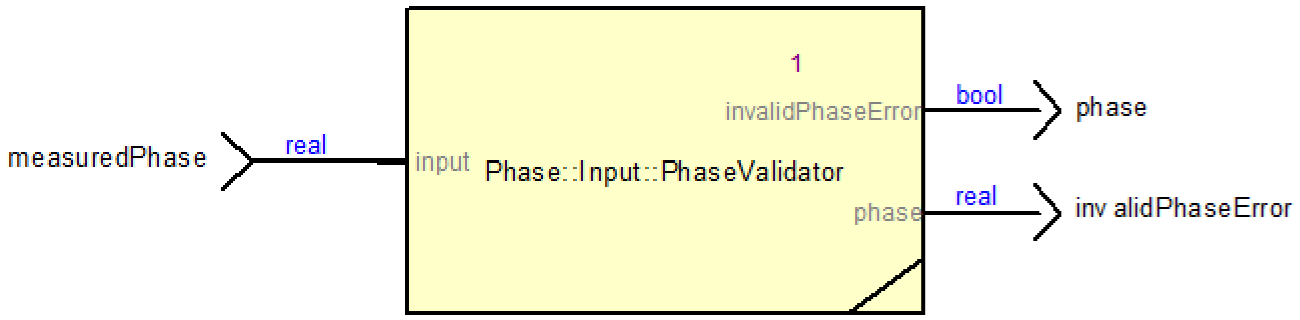
7.4 Collection

7.4.1 Scenario Description

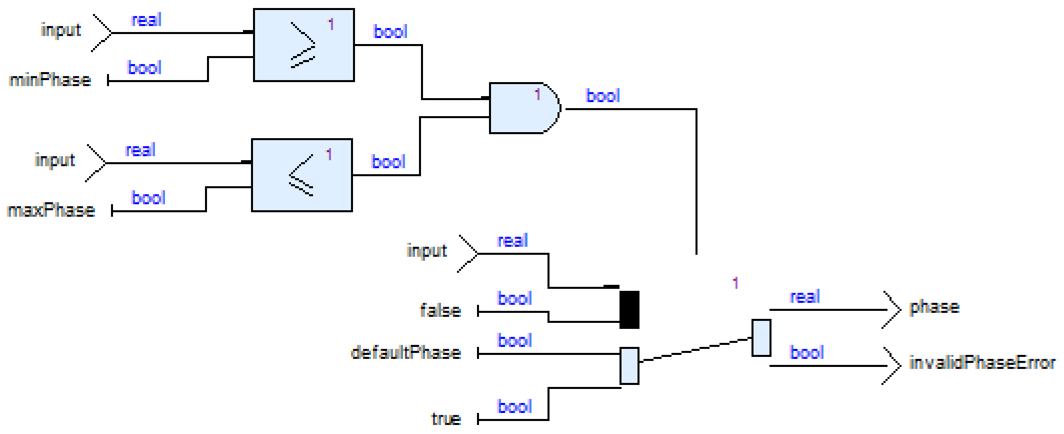
A system can have features that affect more than one components at the same time. If they all share the same type and require identical manipulation, then it makes sense to group them and manage them as a collection. If that was the choice taken during modeling, as explained in section 6.4, then it is a good idea to mirror that approach at the implementation level.

A method to deal with components of the system as a group should have the following characteristics:

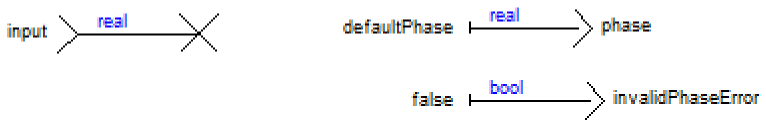
- The implementation should stay the same regardless of the exact number of items in the collection.
- It must be clear how each element of the collection is manipulated.
- It should reduce manual work as much as possible.



(a) Calling Operator



(b) Concrete Operator



(c) Mock Operator

Figure 7.5: SCADE Optional Node

7.4.2 Suggested Pattern

Handling this scenario in SCADE is very easy, as it provides support to deal with aggregate elements via arrays. The only caveat is that it does not contain traditional control flow statements like loops. Instead, collections must be manipulated via higher-order functions like maps and folds. However, once these functions are in place, the implementation because completely stable; changing the number of items only requires updating the constant that holds this value, and this process could be automated via a script or a similar approach.

Discussion

With this pattern, it is possible to handle a variable number of elements with minimal impact to the implementation. However, before using it there are two things to keep in mind. First, it requires using higher-order functions to manipulate the elements in the collection. Second, SCADE does not allow empty collections; if a variant could end up with an empty array, then this approach would need to be combined with the optional node pattern, described in section 7.3.

7.5 Summary

This chapter introduced a series of patterns that can help the implementation in SCADE Suite to be compatible with the modeling approach previously described. The first describes how to provide alternative component implementations when the signature remains unchanged. The second, on the other hand, presents a couple of approaches to deal with alternative components with different signatures; one of them has not been implemented, but highlights the need of an abstract struct. Next, the chapter presented an approach for an optional node. Finally, it described the collection pattern which is useful when it is possible to have one or more components of the same type.

Chapter 8

Evaluation

8.1 FCU

The Fuel Control Unit (FCU) is a part of a turbo engine that determines the rate and the amount of fuel required to meet the power demanded by the pilot and the engine controller [17, 28]. As part of its operation, the FCU will receive a fuel request from other parts of the engine control. The unit will limit the value, determine the best schedule to use to provide steady performance, and will command the hardware as necessary to satisfy that request. Also, it will meter the fuel flow and control the shut-off valve to turn off the engine.

Although the operation of the FCU should be the same for all engines, in practice it varies a lot from model to model. There are multiple combinations of equipment that can be used to implement the unit; different types of sensors, valves, controls, among others. The control, fault management, inputs, and outputs will change with each hardware difference, so the software cannot be the uniform for all instances; it must adapt to manage the unit appropriately. Therefore, an FCU is a good choice to evaluate the patterns presented in this work since it is a relatively small component with a considerable amount of variability.

8.2 Evaluation

The evaluation of these patterns consisted in the creation of a family architecture model that was later used to accommodate five real engine instances. Out of them, four are turboshaft engines, used on helicopters of different vendors and sizes, and one is a turbofan engine commonly installed in medium-size business jets. Even though all of them share similar operation principles, they still have a considerable amount of variability; they employ various equipment, and provide different features and interfaces (as necessary by the particular airframe maker they support).

The model focuses on the fuel control unit, but also includes other parts necessary to describe this component appropriately. It contains the top-level engine decomposition, the fuel system hardware, communication buses, some sections of the engine electronic control (EEC) and the FCU per se. To provide some idea of its scope, the final model provided support for 19 features, contained 50 system requirements, 97 blocks, 107 interface blocks, and 40 diagrams in total.

8.3 Results

The methodology described in this document allowed the model to accommodate five real aircraft engine instances. However, it is important to note that to do so, some effort was necessary to standardize interfaces, components, and properties. For instance, the fault accommodation mechanism for each engine was quite different from each other. Each engine approach had to be reworked so that the vocabulary and requirements became consistent with each other; the patterns were applicable only after this standardization. In other words, the patterns cannot be applied without performing the rest of the methodology.

Unfortunately, there was no baseline to contrast the performance of this methodology with other approaches. It appears that the effort of modeling the family architecture is comparable to developing a single system. However, informally, it seems that modeling and maintaining the instance models is indeed more efficient than cloning or creating them from scratch; instead of duplicating the model, only a handful of blocks required modification. On average, only 12 blocks needed changes. Seven interface blocks had to be adapted to leave interfaces in a consistent state, and only two activity diagrams had to be added to describe customization points. Table 8.1 shows a summary of the effort necessary for each model.

The following section presents the results of using each pattern in practice.

Behavioral Patterns

1. **Variability in Activity Diagrams**
   The initial attempt was to create structural diagrams directly from the requirements.

	Blocks	Interface Blocks	Flow Properties	Redefined Properties	Redefined Flows	Activity Diagrams
Family	60	101	309	—	—	14
Instances	12	8	0	36	13	2

Table 8.1: Results Statistics

Although, they were mostly extracted from real engine specification documents and supposedly provided a complete understanding of the system, we found several ambiguities; even experts we worked with had trouble understanding them. The problem was that some requirements could be interpreted differently, so it was hard to specify accurately what the functionality of each engine was, and how it varied between instances.

We found that the patterns defined in section 5.1, and activity diagrams in general, were extremely useful to solve this problem. They allowed us to get a clear picture of the actual functionality, the inputs and outputs necessary to accomplish it, and showed precisely where variation happens and what its scope is. With this information creating structural diagrams became straightforward, as the decomposition could be extracted almost directly from activities. It is important to highlight that the FCU example mostly had flow-based behavior; in other situations other behavioral diagrams might be better suited to aid in the structural decomposition.

In this example, out of 40 diagrams, 14 were activity diagrams. In them, two activities were tagged as *customization points*, and six conditional nodes were used to encode variability. The resulting diagrams were readable, and variability scope was easy to identify. However, there were no complex cases, like a single diagram with many conditional nodes or nested variability (like a conditional node inside another conditional node). Cases like these could make diagrams harder to read; refactoring the models or additional tool support could be used to address this problem.

2. **Variability in Sequence Diagrams**
   In the example, one sequence diagram was necessary. It was useful to show the interaction between the system and the external entities, which in this case were the pilot, the airplane avionics, and the remote channel. The diagram displayed the communication between the different actors and required using the combined fragment construct to introduce variability, as described in section 5.2, only once. As with activity diagrams, complex cases that require nesting of combined fragments could become hard to understand; refactoring of the models, like creating sub-diagrams, and tool support can potentially address this problem.

   From this example, it might appear that activity diagrams are more important than sequence diagrams. However, this is not the case. Part of the behavior of the FCU that could have been described with them was already present in activity diagrams. Nevertheless, other components that have a more heavily message oriented behavior might require using more sequence diagrams.

Structural Patterns

1. **Variable Property Pattern**
   The variable property pattern was very useful during the development of the model to specify block properties that vary from instance to instance. Out of the three ways to deal with quantitative variability mentioned in section 6.1, only the *PDI* approach was used, but the *Value Properties* strategy was also valid; choosing between these two appears to depend only on when the data will be provided. On the other hand, the *intervals* approach was not necessary within this model. It is very likely that if test cases were considered, this method would have been used too.

2. **Optional Component**
   This example presented the optional component scenario twice. In both cases, the mock component approach was used since the aggregate block could work correctly even with default values. Although the component removal pattern was suitable too, there was no need to disturb the rest of the model by deleting the optional components altogether.

   In the end, it appears that the decision of which of these two patterns to choose should mirror the implementation approach that will be used. If the implementation can work properly with default values, then the mock component seems more appropriate, since it can leave the rest of the code untouched. Analogously, if it is possible to refactor interfaces for each instance or if there are cases when default values are not adequate, then the component removal pattern should be used.

3. **Alternative Component Pattern**
   The FCU model made use of the alternative components pattern presented in section 6.3 seven times. Since the implementation of two of them had to be done on a per-instance basis, they were marked as customization points. For the other five the alternatives were known at the family architecture model, so they were provided explicitly at that level. All of them had two options to choose from, but more could have been available if needed.

   Although the implementation of this pattern was straightforward, careful evaluation with experts from our industry partner showed that this pattern can become problematic in some cases. The issue is that as a product line evolves, new alternatives can be introduced. When that happens, it is possible that new flows will be added too. If this process is not performed with care there can be interface duplication or inconsistencies, which can lead to model deterioration. This problem can be mitigated to some extent with good practices, like data dictionary and careful design.

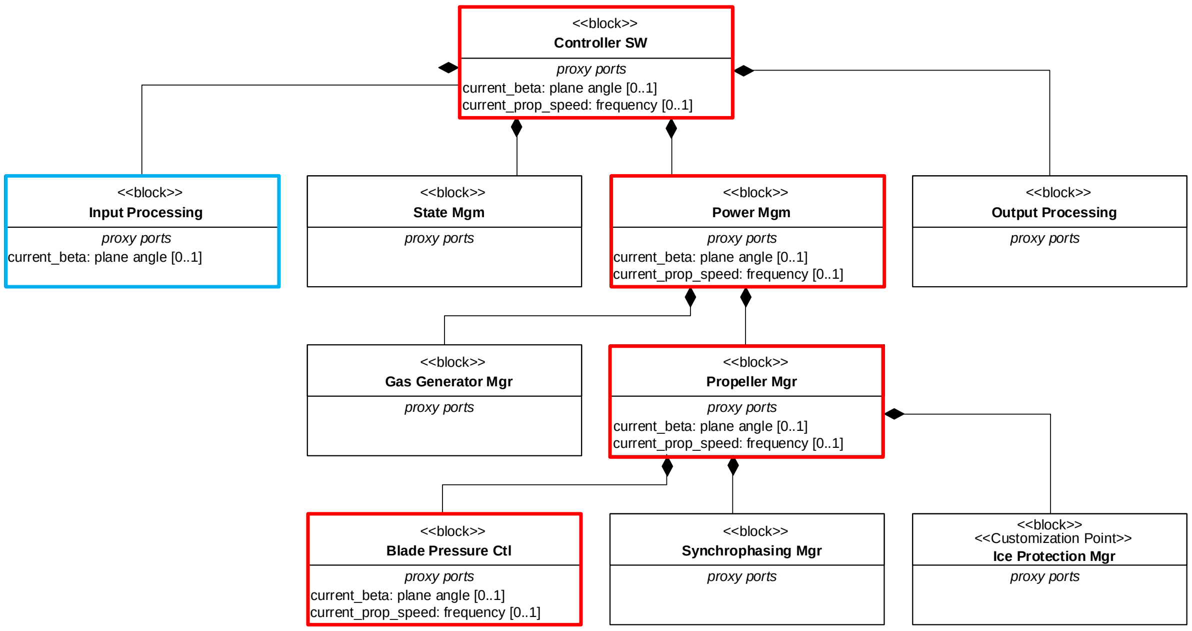


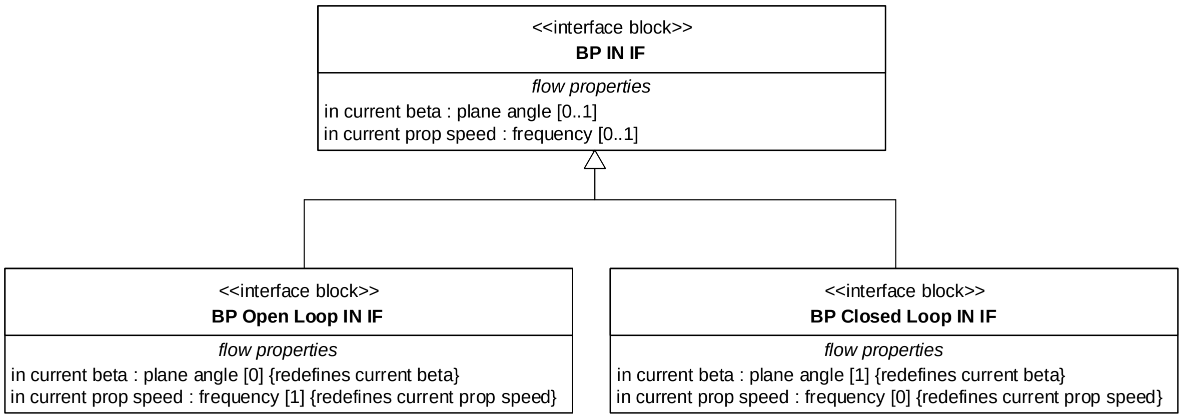
Figure 8.1: Interface changes impact without Data Flow Organization Pattern

However, a study about the evolution of models with variability can provide valuable information about how to solve the problem.

4. **Collection Pattern**
Although the pattern was not applied on the FCU, engineers at the industry partner identified potential uses in other parts of an engine controller, in particular parts that need to be repeated per stage of a turbine or a compressor. However, since the number of stages is limited (typically less than 10), this variability can also be implemented using the OR component pattern, with one OR component per stage.

5. **Data Flow Organization Pattern**

This pattern was crucial to keep the interfaces in the FCU case study under control, and to make other patterns practical and efficient. Organizing flow properties in interface blocks allowed several connectors to be represented in the models as only one; this makes diagrams more readable as connectors are not overlapping each other. The example had 60 parent interface blocks (not counting inherited blocks), with an average of 5.15 flow properties; without this pattern there would have been 309



<<interface block>>
BP IN IF
flow properties
in current beta : plane angle [0..1]
in current prop speed : frequency [0..1]
<<interface block>>
BP Open Loop IN IF
flow properties
in current beta : plane angle [0] {redefines current beta}
in current prop speed : frequency [1] {redefines current prop speed}
<<interface block>>
BP Closed Loop IN IF
flow properties
in current beta : plane angle [1] {redefines current beta}
in current prop speed : frequency [0] {redefines current prop speed}

(a) Interface Block

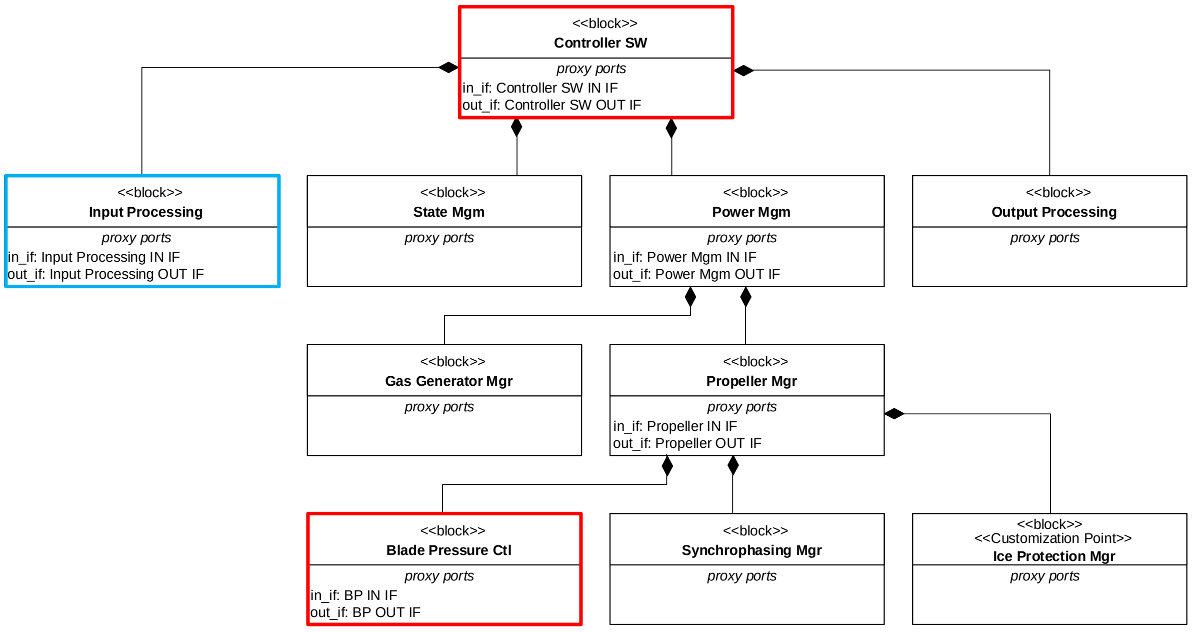


Figure 8.2: Interface changes impact with Data Flow Organization Pattern

connectors. Moreover, adopting this pattern allowed reducing the changes necessary to accommodate interface updates. Without it, all the intermediate interface blocks would have been changed too. In this example, eight modifications would have been needed per flow property in the worst case. Other systems with deeper component nesting might have even harsher side-effects.

Figures 8.1 and 8.2 show the impact necessary to accommodate interface changes in the propeller case study. In the former, the model is not using the data flow organization pattern, so all intermediate nodes require modifications every time *Blade Pressure Ctl* changes; this impact is shown in red. The latter does use the pattern, so only two updates are required (the block in blue also requires modification if the *Equipment Differences Pattern* is used).

6. **Bus Alternative Pattern**
   The bus alternative pattern was used four times as that is the number of buses present in the system. Unfortunately, the approach could not be exercised entirely since not all possible variations described in section 6.6.2 were present. In particular, the *Bus Change* type was not applied since all instances made use of only one protocol per bus. However, the other two, *data* and *recipient differences* were used several times; they were able to isolate the system from communication related variability.

7. **Equipment Differences Pattern**
   The FCU described in this example can have seven pieces of equipment that can vary between instances; five are optional, and the other two can provide alternative choices. In the original implementations, the effects of these variations scatter through many parts of the system. Using the pattern described in section 6.7, the system was successfully isolated from the input/output differences caused by equipment changes. The internal controller structure relies on standard flows, and only the blocks in the external layers and their corresponding interface blocks required modifications. However, this protection was limited to flows; variations that demanded internal logic changes could not be prevented, and required using other patterns.

8.4 Summary

This chapter demonstrates the ability of the proposed methodology to encode and handle variability through the fuel control unit (FCU) case study. In this example, the family architecture model was able to accommodate five real aircraft engine instances. Furthermore, the chapter provides results and experiences about each of the patterns described in this work.

Chapter 9

Related Work

9.1 Variability Modeling

This section describes the previous work focused on modeling variability, especially in UML and SysML.

The Common Variability Language (CVL) is a standard created and maintained by the Object Management Group for the specification and resolution of variability [23]. One of its biggest advantages is that it allows specifying variability into existing models as long as the language is compliant with the MOF-metamodel; this includes any UML-derived language, such as SysML. There are some similarities between CVL concepts and the methodology in our work. First, the variability and resolution models are equivalent to the feature and instance models we define in Clafer. Likewise, the base model is reminiscent of the family architecture model, while the resolved models are similar to the instance architecture models. However, there are also three big differences. First, our methodology takes advantage of existing SysML constructs; since CVL is a more generic language it does not leverage them. Second, CVL allows adding variability to existing models. However, it might not be efficient or possible to do so if they have not been designed to deal with it from the start. Our methodology on the other hand forces to think about variability from the beginning. Finally, CVL would create a variant model that is detached from the base model; our methodology defines variant models as subclasses of the base model. To get that effect one would have to create a set of instance level subclasses and then annotate them in CVL fashion.

Trujillo et. al. [44] describes the approach followed to model a wind turbine system in SysML. Although their case study is similar to the ones presented in this work, their solution is quite different to ours. Theirs focuses on defining a set of stereotypes for many constructs in SysML; they have one for blocks called BlockVariationPoint, for parts ObjectVariationPoint, among others. It is unclear if they use any other construct in SysML to control variation. Furthermore, it is not stated if and how they trace the variation points back to the feature model. Clauss [10] introduced a similar approach, but using UML.

Ziadi et. al. [49] shows an approach very similar to the behavioral patterns included in our work. Furthermore, they also make use of constraints to ensure the consistency of the derived instances. Despite these similarities, their work is restricted to UML, and they do not link the stereotypes to the feature model.

Bayer et. al. [6] represents the most similar approach to the one presented in our work. They also make use of specialization and redefinition to tackle variability, and enhance these mechanisms with stereotypes. Moreover, they analyze how variations can affect the implementation code, and how they can put the models in practice. However, their work is restricted to UML, considers only class diagrams, and does not provide any of the patterns presented in our work.

Bachman et. al. [4] present a different way of modeling variations in UML. They introduce a more intricate metamodel and propose using it to create a specialized variability view. Since this approach requires an external representation, it is more reminiscent of the feature model than the methods presented in our work.

Finally, there also have been attempts to show variability in other types of models. For instance, Zaid and De Troyer [48] showed a new way to model variability in data to be stored in a database. Although this kind of approaches are worth considering, their target applications are outside the scope of this work.

9.2 Avionics Systems Modeling

Since avionics applications are highly complex, it is not surprising that there have been several attempts to model their architectures.

Quadri et. al. [37] present the results of a multi-year project aimed at unifying SysML and MARTE for the description of real-time and embedded avionics systems. It is important to mention that their work is ambitious; it defines a methodology, ways to model systems, and how to put them in practice. However, their approach does not deal with variability in any way.

Behjati et. al [7] realized the usefulness of AADL and tried to extend SysML to include concepts of this language. They accomplished this by extending SysML with a profile they developed, called ExSAM. The feasibility of this approach was shown with a large-scale case study. However, this approach does not cover variability.

9.3 Summary

This chapter describes how the methodology and patterns described in this thesis relate to previous endeavors. In particular, it compares our approach with existing variability modeling works. In this regard, our proposed methodology differs from others since it takes advantage of SysML constructs and adapts them to the avionics domain needs. Also, this chapter compares our methodology to previous attempts to model avionics systems.

Chapter 10

Conclusion

10.1 Summary of Results

This thesis presented an approach for modeling avionics applications with variability. First, the study evaluated SysML and concluded that it is indeed fit for encoding variability in models. However, it needs to be extended with a profile such as the one included in Appendix B. Next, this work introduced a methodology and a series of structural and behavioral patterns to model this kind of systems while keep variability under control. Implementation patterns were also included to show how to put the model into practice. Finally, this whole approach was evaluated via a case study based on five real engine instances.

10.2 Threats to Validity

There are two circumstances that could affect the validity of this work. First, some of the tool features previously described, like the interface compatibility analysis in section 6.5, were not implemented. Although it is very likely that this functionality in the tools could simplify modeling and development tasks, without proper evaluation it is not possible to ensure how effective they are.

Second, the methodology was evaluated in the context of airplane engine control, which is a particular case of avionic system: it is relatively closed system, concerned mostly with continuous control, and extremely safety-critical (the entire system is classified as DAL

A). There is a wide range of avionics systems on an aircraft, some are similarly critical and involving continuous control, such as flight guidance; others may be more data-driven, such as flight management systems, and of lower criticality levels, such as entertainment system. The SysML patterns are likely applicable to most avionics systems; SCADE is less appropriate for data intensive systems, however.

10.3 Future Work

There are three possible ways to extend this work. First, our current approach did not take into account timing while dealing with variability. This is because SysML does not provide a comprehensive way to model this information. However, MARTE can be used to add this functionality. It would be quite useful to explore how to encode variability when this profile is used, in particular with regards to timing.

Similarly, this work did not focus on parametric diagrams, which can be used to specify system properties to perform advanced engineering analysis. If variability is encoded and handled properly, system properties can be stored at the family architecture level, and the parametric diagrams can then be automatically adapted to each instance. This information could be useful to ensure that each variant behaves according to expectations, or can even provide the ability to execute design exploration.

Finally, the scope of the case study can be increased to include the whole engine. That way there can be even more certainty of the validity of the presented approach.

APPENDICES

Appendix A

Introduction to SysML

This appendix contains a small introduction of some of the SysML diagrams and features used throughout this thesis. Please refer to the current SysML specification [32] for a more in-depth explanation.

A.1 Diagrams

‘

In SysML, a model consists of a series of diagrams representing different aspects of a system. Each of them is enclosed by a diagram frame, which contains a header, a content area, and a description, as shown in figure A.1.

- **Header:** Located at the top left side of the frame, this section includes information useful to identify the current diagram, such as the diagram kind, diagram name, among others.
- **Content Area:** Also referred to as canvas, this section contains the graphical representation of the elements in the diagram.
- **Description:** This is an optional element that provides additional information about the diagram.

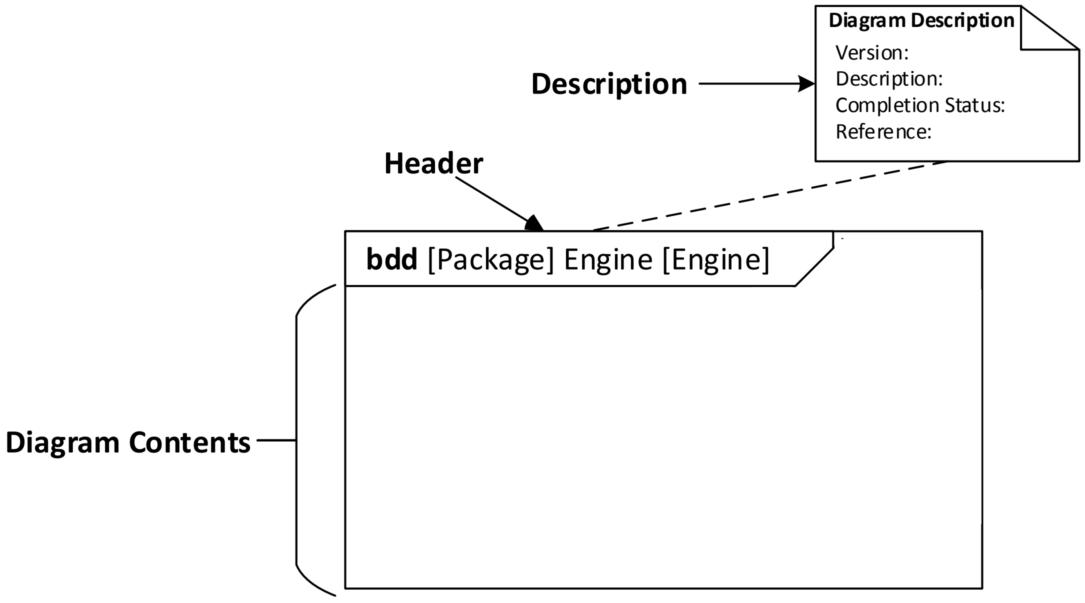


Figure A.1: Diagram Frame

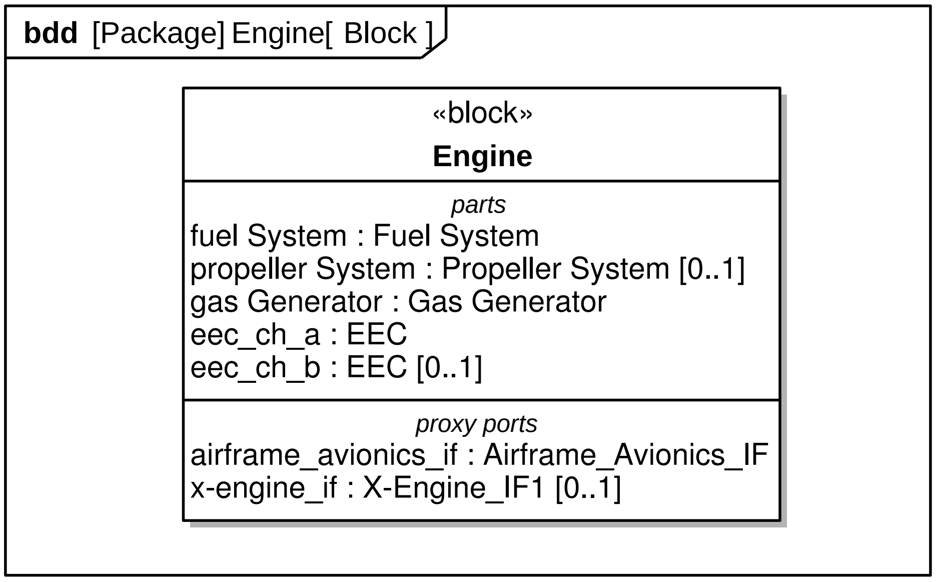

bdd [Package] Engine[Block]
«block»
Engine
parts
fuel System : Fuel System
propeller System : Propeller System [0..1]
gas Generator : Gas Generator
eec_ch_a : EEC
eec_ch_b : EEC [0..1]
proxy ports
airframe_avionics_if : Airframe_Avionics_IF
x-engine_if : X-Engine_IF1 [0..1]

Figure A.2: Block Example

A.2 Blocks

Blocks in SysML are modular units of system description. Each block defines a collection of features, which can be structural or behavioral in nature, to specify a system, a part, or another element of interest. In other words, a block can represent quite different concepts relevant in the system, like software and hardware components, human elements, among others. One important note is that blocks allow reuse; they can be used in multiple contexts.

As shown in figure A.2, a block is depicted as a rectangle and is identified as such with the bracket symbol «block», called guillemets, at the top of the element. It contains several compartments; the one at the top is mandatory, and always contains its name; the rest are optional, and describe block characteristics, like properties, operations, and constraints.

Properties are one of the structural features that can appear in a block. The model displays them with their name, type and extra attributes, like multiplicity (shown within brackets). There are many types of properties, as described below:

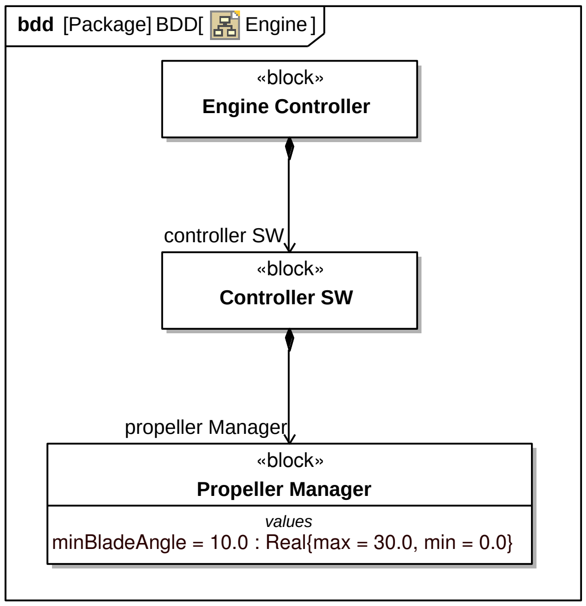
- **Part Property:** Represents the local usage of a block in the context of the enclosing block.
- **Reference Property:** It is a part used, but not owned by the enclosing block; it is not composition.
- **Value Property:** Provides a quantifiable attribute; is typed by a value type.

A.3 Structural Diagrams

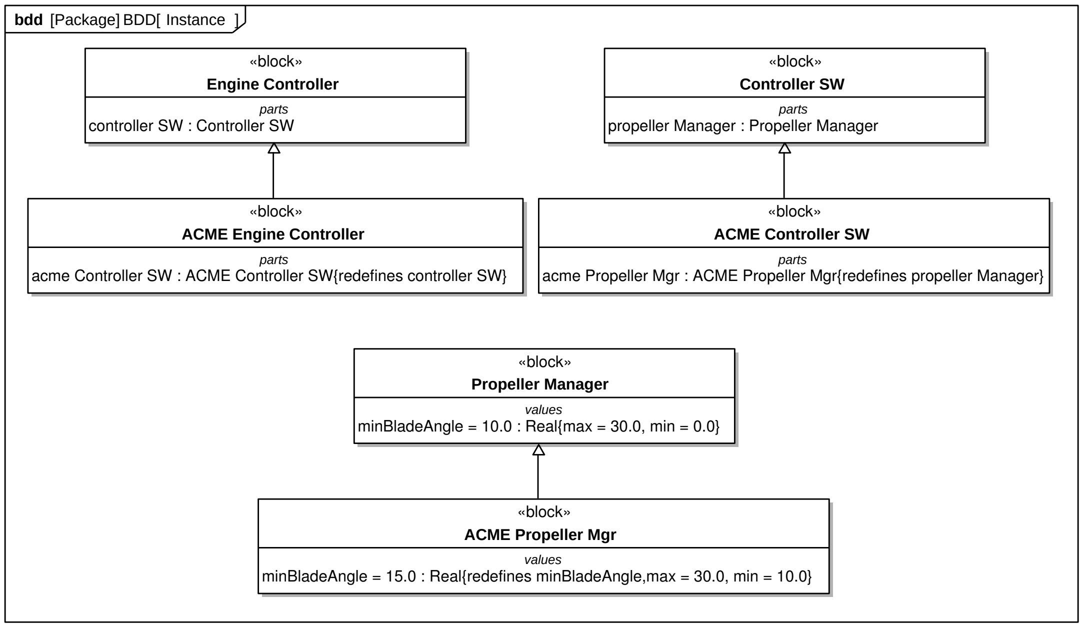
A.3.1 Block Definition Diagram (BDD)

The block definition diagram (BDD) is useful to depict blocks, in terms of their features, and the relationships between them, such as associations, generalizations, and dependencies. Furthermore, it can also be used to specify instances of blocks.

One important type of relationship is generalization. In it, one or more blocks derive from a more general one referred to as the parent block. With this construct, it is possible for the parent to define a structure that is inherited by the children. However, the children are free to add features, or even modify the existing ones via redefinition; this allows changing the property to something more adequate for that child.



(a) Engine Controller BDD


bdd [Package] BDD[Instance]
«block»
Engine Controller
parts
controller SW : Controller SW
«block»
Controller SW
parts
propeller Manager : Propeller Manager
«block»
ACME Engine Controller
parts
acme Controller SW : ACME Controller SW{redefines controller SW}
«block»
ACME Controller SW
parts
acme Propeller Mgr : ACME Propeller Mgr{redefines propeller Manager}
«block»
Propeller Manager
values
minBladeAngle = 10.0 : Real{max = 30.0, min = 0.0}
«block»
ACME Propeller Mgr
values
minBladeAngle = 15.0 : Real{redefines minBladeAngle,max = 30.0, min = 10.0}

(b) Redefinition without Bound Reference

Figure A.3: Redefinition

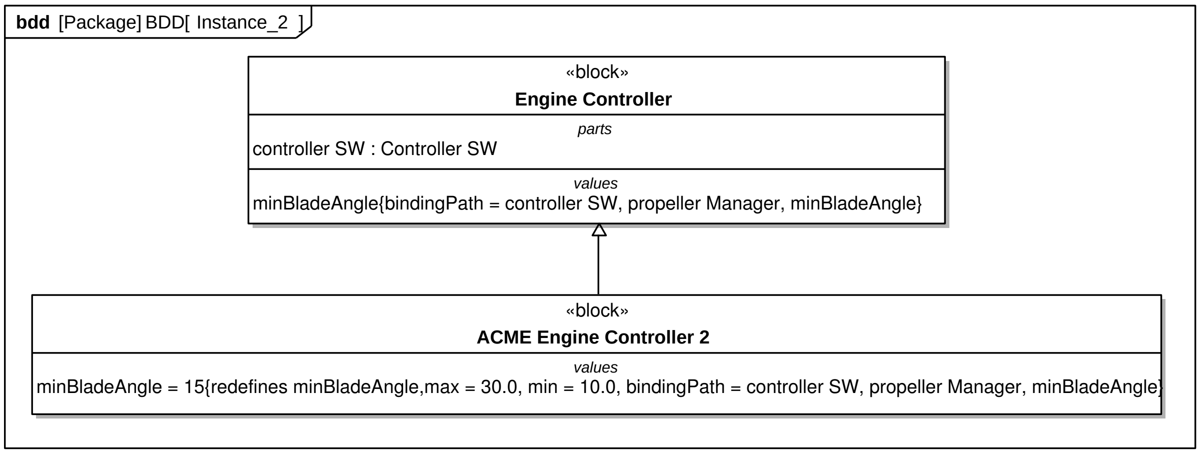

bdd [Package] BDD[Instance_2]
«block»
Engine Controller
parts
controller SW : Controller SW
values
minBladeAngle{bindingPath = controller SW, propeller Manager, minBladeAngle}
«block»
ACME Engine Controller 2
values
minBladeAngle = 15{redefines minBladeAngle,max = 30.0, min = 10.0, bindingPath = controller SW, propeller Manager, minBladeAngle}

Figure A.4: Redefinition with Bound Reference

One important aspect to highlight is that a block can only redefine the features it contains directly; the properties of its parts can only be modified by successors of those parts themselves. In figure A.3, for example, *Engine Controller* cannot redefine the *minBladeAngle* property of *Propeller Manager*. To make this change, the intermediate blocks must be subclassed until *Propeller Manager* block is reached. Since this process can be very time-consuming, SysML 1.4 [34] will introduce a construct called bound-references. With it, SysML will allow an enclosing block to access and redefine one of the features of a part as shown in figure A.4. Note that the bound reference can be defined in a BDD or an IBD as necessary.

A.3.2 Internal Block Diagram (IBD)

The internal block diagram (IBD) displays the internal structure of a block by showing how its parts connect with each other. In particular, an IBD contains a graphical representation for parts, references, and values, and shows how they are linked via connectors and ports.

Ports are of great importance, since they can specify the flow of inputs and outputs, messages, and operations between the endpoints of the connector. An IBD displays them as squares (or rectangles) in the boundary of the owning block. They are decorated with an arrow representing the direction (in, out, in/out) of the flows within a port; this only happens if they all follow the same direction. Note that any item can flow through a port regardless if it is data, material, or energy, and a port can support more than one type.

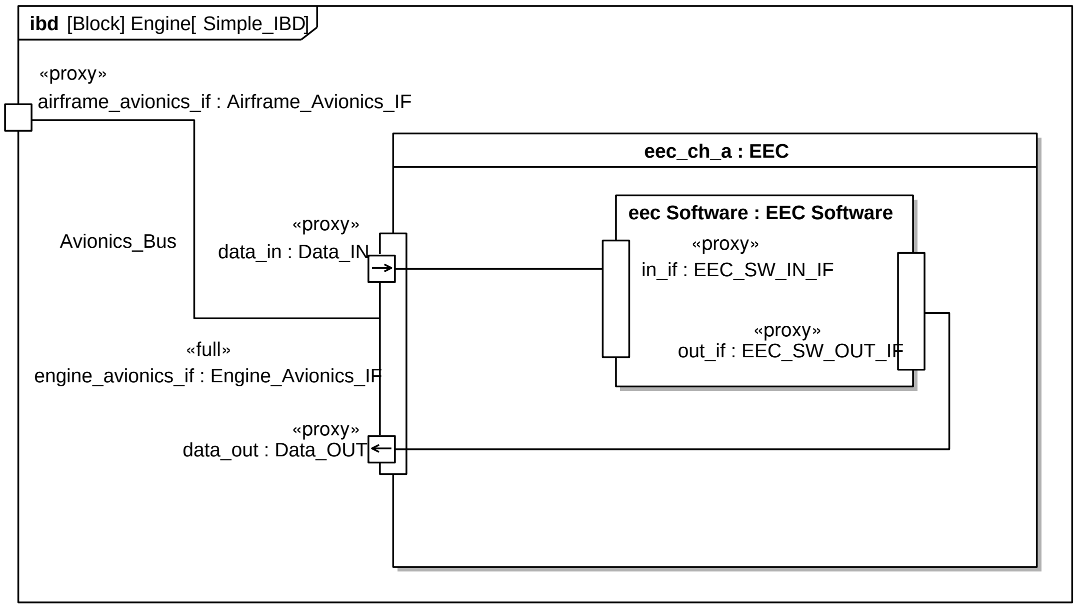


Figure A.5: IBD Example

It is important to highlight that ports have suffered drastic changes between SysML releases. Version 1.2 made use of standard and flow ports. Version 1.3, on the other hand, deprecated those two types and replaced them with full and proxy ports; they are described below since these are the ones used throughout this work.

- **Proxy Ports:** They expose features of either the owning block or its internal parts, but the ports themselves are abstract; they do not represent real parts. They are typed by an interface block and cannot contain internal structure.

- **Full Ports:** They represent a part in the owning block. They are typed by blocks and can contain internal structure.

Note that SysML differentiate these two types via the guillemets on top of them. Also, both of them allow nesting, which means that a port can contain another port. This case is presented in figure A.5, where *data_ in* and *data_ out* are nested in the *engine_ avionics_ if* port.

Once ports are in place, a connector can be used to establish some link between them. Although this construct does not present detailed information about the connection per se, more details can be presented by typing it with an association block.

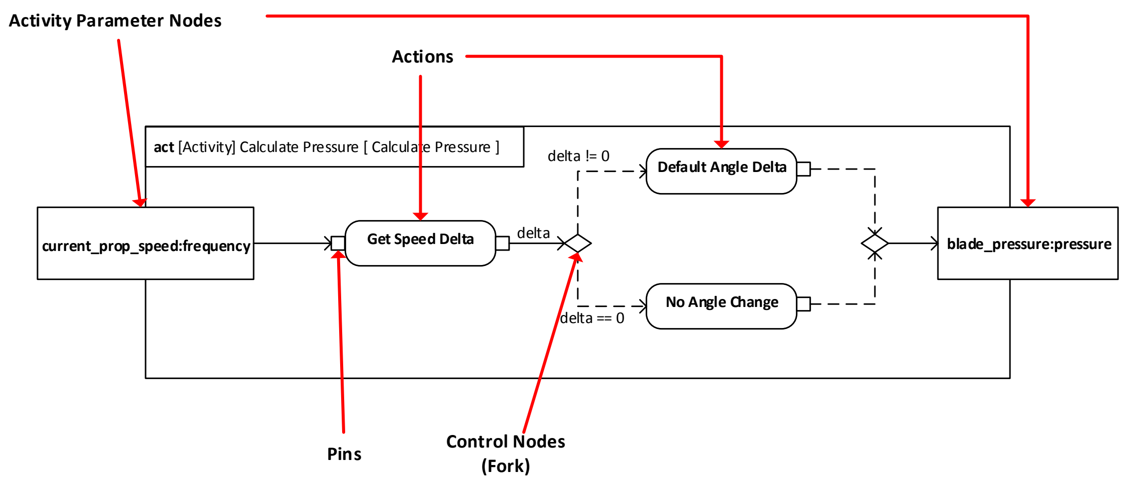


Figure A.6: Activity Diagram Example

A.4 Behavioral Diagrams

A.4.1 Activity Diagram

SysML introduces activity diagrams to show flow-based behavior and describe how the provided inputs are turned into the expected outputs. Although similar to the widely-used functional flow diagrams, activities provide additional functionality, like allowing to link behavior to structural elements and the ability to model continuous flows.

From the notation point of view, an activity diagram describes a single activity. It specifies the interface via parameter nodes, which are represented as boxes in the border of the diagram frame; each of them contains a name and a type. Similarly, the internal structure is decomposed into a group of actions, each of which is presented as a rounded box, with pins standing for inputs and outputs. Furthermore, nodes are used to express the order of execution; they can specify the start and end of the activity, where control forks, among others. Note that solid arrows represent the flow of objects, while the dashed ones specify the flow of control.

Figure A.6 presents an example with the notation used in activity diagrams.

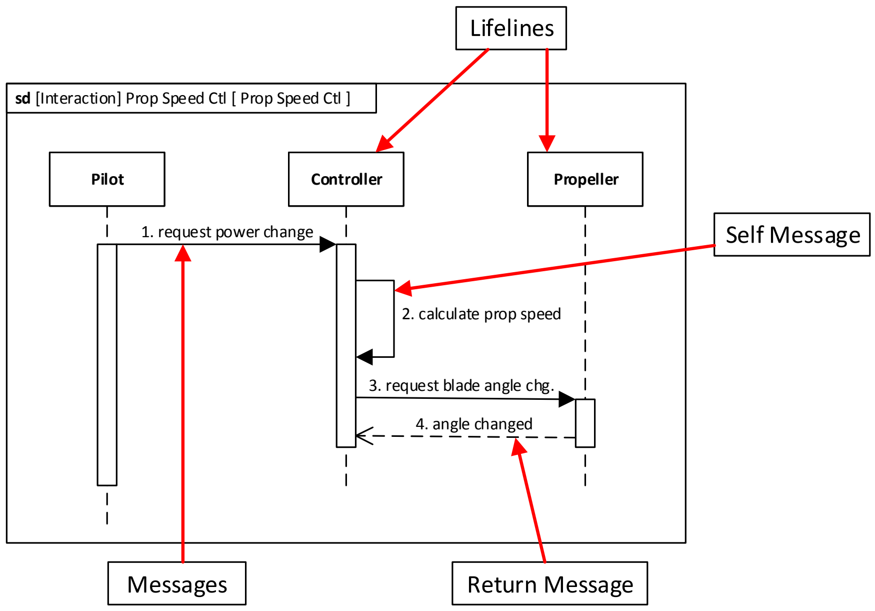


Figure A.7: Activity Diagram Example

A.4.2 Sequence Diagram

SysML makes use of sequence diagrams to allow the specification of message-based interactions. Since the diagram was not modified, it retains all the constructs and functionality as in UML. In them, each item is represented via a lifeline, which can be an actor, a part of the system, or another system altogether; it can be any entity participating in the interaction. Furthermore, messages represent the invocation of a service, sending a signal, or some other type of control flow structure. What is important is that they are presented in chronological order.

Figure A.7 presents an example showing the notation of this diagram.

A.5 Extending SysML via Profiles

A profile is the standard way to extend UML, and in consequence SysML. In fact, SysML itself and other languages like MARTE are also created as profiles. The primary extension construct is the stereotype, which allows augmentation in two ways. First, it allows creating new language constructs by overloading existing concepts, like metaclasses or other stereotypes. The second way is to include additional information to existing types via annotations; these do not collide with or affect existing SysML data in any way. Note that since these are extensions, any tool that supports UML should work with or without profiles.

As an example, figure B.1 in appendix B presents the profile used to enhance SysML with all the product line features described throughout this work.

Appendix B

Variability SysML Profile

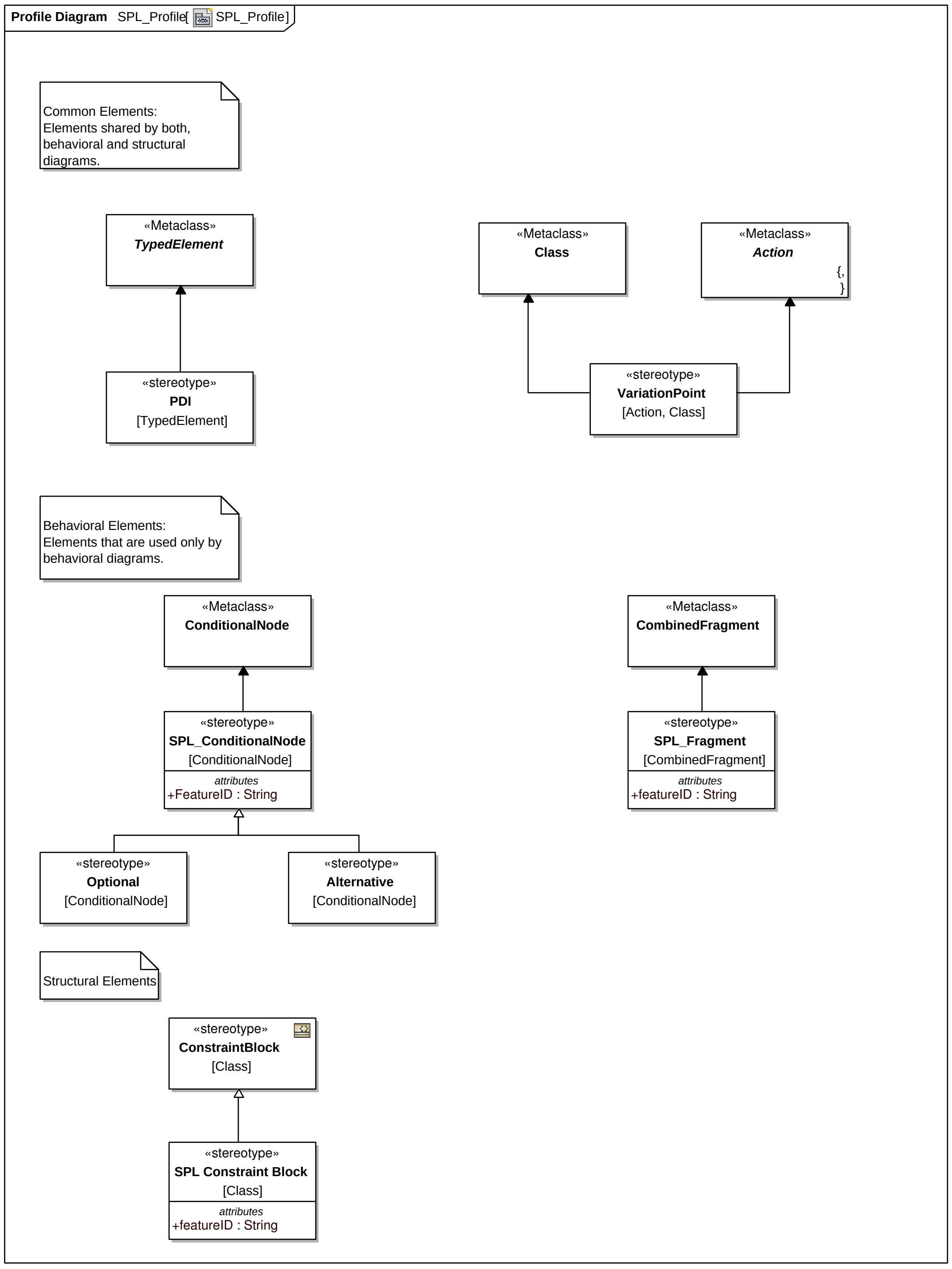

Profile Diagram SPL_Profile[SPL_Profile]
Common Elements:
Elements shared by both, behavioral and structural diagrams.
«Metaclass» TypedElement
«stereotype» PDI [TypedElement]
«Metaclass» Class
«Metaclass» Action {, }
«stereotype» VariationPoint [Action, Class]
Behavioral Elements:
Elements that are used only by behavioral diagrams.
«Metaclass» ConditionalNode
«Metaclass» CombinedFragment
«stereotype» SPL_ConditionalNode [ConditionalNode]
attributes
+FeatureID : String
«stereotype» SPL_Fragment [CombinedFragment]
attributes
+featureID : String
«stereotype» Optional [ConditionalNode]
«stereotype» Alternative [ConditionalNode]
Structural Elements
«stereotype» ConstraintBlock [Class]
«stereotype» SPL Constraint Block [Class]
attributes
+featureID : String

Figure B.1: SPL Profile

References

[1] SCADE Suite KCG C Code Generator, 2014.

[2] Mark ARINC. Digital Information Transfer System (DITS) Part 1, 1995.

[3] SAE ARP4754. Certification Considerations for Highly-Integrated or Complex Aircraft Systems. *SAE, Warrendale, PA*, 1996.

[4] Felix Bachmann, Michael Goedicke, Julio Leite, Robert Nord, Klaus Pohl, Balasubramaniam Ramesh, and Alexander Vilbig. A Meta-model for Representing Variability in Product Family Development. In *Software Product-Family Engineering*, pages 66–80. Springer, 2004.

[5] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In *3rd International Conference on Software Language Engineering*, Eindhoven, The Netherlands, 10/2010 2010.

[6] Joachim Bayer, Sebastien Gerard, Øystein Haugen, Jason Mansell, Birger Møller-Pedersen, Jon Oldevik, Patrick Tessier, Jean-Philippe Thibault, and Tanya Widen. Consolidated Product Line Variability Modeling. In *Software Product Lines*, pages 195–241. Springer, 2006.

[7] Razieh Behjati, Tao Yue, Shiva Nejati, Lionel Briand, and Bran Selic. Extending SysML with AADL Concepts for Comprehensive System Architecture Modeling. In *Modelling Foundations and Applications*, pages 236–252. Springer, 2011.

[8] Gary Chastek, Patrick Donohoe, Kyo Chul Kang, and Steffen Thiel. Product Line Analysis: a Practical Introduction. Technical report, DTIC Document, 2001.

[9] J Chism. Overview and Status of the Object Oriented System Engineering Methodology (OOSEM). *INCOSE Insight*, 7(2):31–33, 2004.

[10] Matthias Clauß. Generic Modeling Using UML Extensions for Variability. In *Workshop on Domain Specific Visual Languages at OOPSLA*, volume 2001, 2001.

[11] Krzysztof Czarnecki and Michał Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In Robert Glück and Michael Lowry, editors, *ACM SIGSOFT/SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes in Computer Science*, pages 422 – 437, Tallinn, Estonia, 2005. Springer-Verlag, Springer-Verlag.

[12] RTCA Do. 178B: Software Considerations in Airborne Systems and Equipment Certification. *December, 1st*, 1992.

[13] RTCA Do. 178C: Software Considerations in Airborne Systems and Equipment Certification. *January, 5th*, 2012.

[14] Huascar Espinoza, Daniela Cancila, Bran Selic, and Sébastien Gérard. Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems. In *Model Driven Architecture-Foundations and Applications*, pages 98–113. Springer, 2009.

[15] Jeff A Estefan et al. Survey of Model-Based Systems Engineering (MBSE) Methodologies. *Incose MBSE Focus Group*, 25:1–70, 2007.

[16] Esterel Technologies. Esterel Technologies, 2014.

[17] FAA. Aviation Maintenance Technician Handbook—Powerplant: FAA-H-8083-32, Volume II. *US Department of Transportation.*

[18] Federal Aviation Administration. *Pilot's Handbook of Aeronautical Knowledge.* Skyhorse Publishing Inc., 2009.

[19] Peter H Feiler, David P Gluch, and John J Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, DTIC Document, 2006.

[20] Guillaume Finance. SysML Modelling Language Explained(2010), 2010.

[21] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language.* Elsevier, 2011.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson Education, 1994.

[23] Ø Haugen. Common Variability Language (CVL). *OMG Revised Submission*, 2012.

[24] NS Haverty. MIL-STD 1553-A Standard for Data Communications. *Communication and Broadcasting*, 10:29–33, 1985.

[25] INCOSE. Model Based Systems Engineering, 2014.

[26] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document, 1990.

[27] Fabrice Kordon, Jérôme Hugues, Agusti Canals, and Alain Dohet. *Embedded Systems: Analysis and Modeling with SysML, UML and AADL*. John Wiley & Sons, 2013.

[28] Andreas Linke-Diesinger. *Systems of Commercial Turbofan Engines*. Springer, 2008.

[29] Howard Lykins, Sanford Friedenthal, and Abraham Meilich. Adapting UML for an Object Oriented Systems Engineering Method (OOSEM). In *Proceedings of the Tenth Annual INCOSE Symposium, International Council on Systems Engineering (July 2000), http://www. omg. org/docs/syseng/02-06-11. pdf*, 2000.

[30] OMG. MARTE specification version 1.1. Technical report, Object Management Group, 2011.

[31] OMG. OMG Object Constraint Language (OCL), Version 2.3.1, 2012.

[32] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3. Technical report, Object Management Group, 2012.

[33] OMG. Object Management Group, 2014.

[34] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.4 Beta. Technical report, Object Management Group, 2014.

[35] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. Software Product Line Engineering. *Springer*, 10:3–540, 2005.

[36] Pratt & Whitney Canada. Pratt & Whitney Canada, 2014.

[37] Imran Rafiq Quadri, Etienne Brosse, Ian Gray, Nicholas Matragkas, Leandro Soares Indrusiak, Matteo Rossi, Alessandra Bagnato, and Andrey Sadovykh. MADES FP7 EU project: Effective High Level SysML/MARTE Methodology for Real-Time and

Embedded Avionics Systems. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–8. IEEE, 2012.

[38] SAE. SAE, 2014.

[39] Bran Selic and Sébastien Gérard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Elsevier, 2013.

[40] Cary R Spitzer and Cary Spitzer. *Digital Avionics Handbook*. CRC Press, 2000.

[41] Neil R Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[42] Esterel Technologies. SCADE Suite. http://www.esterel-technologies.com/products/scade-suite/. Accessed: 2014-09-10.

[43] Esterel Technologies. SCADE System. http://www.esterel-technologies.com/products/scade-system/. Accessed: 2014-09-10.

[44] Salvador Trujillo, Jose Miguel Garate, Roberto Erick Lopez-Herrejon, Xabier Mendialdua, Albert Rosado, Alexander Egyed, Charles W Krueger, and Josune De Sosa. Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy. In *Modelling Foundations and Applications*, pages 293–304. Springer, 2010.

[45] OMG UML. 2.4. 1 Superstructure Specification. Technical report, document formal/2011-08-06. Technical report, OMG, 2011.

[46] Carnegie Mellon University. AADL, 2014.

[47] Michel van Tooren and Arvind G. Rao. *Fixed-Wing Civil Transport Aircraft Design*. John Wiley and Sons, Ltd, 2010.

[48] Lamia Abo Zaid and Olga De Troyer. Towards Modeling Data Variability in Software Product Lines. In *Enterprise, Business-Process and Information Systems Modeling*, pages 453–467. Springer, 2011.

[49] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a UML Profile for Software Product Lines. In *Software Product-Family Engineering*, pages 129–139. Springer, 2004.