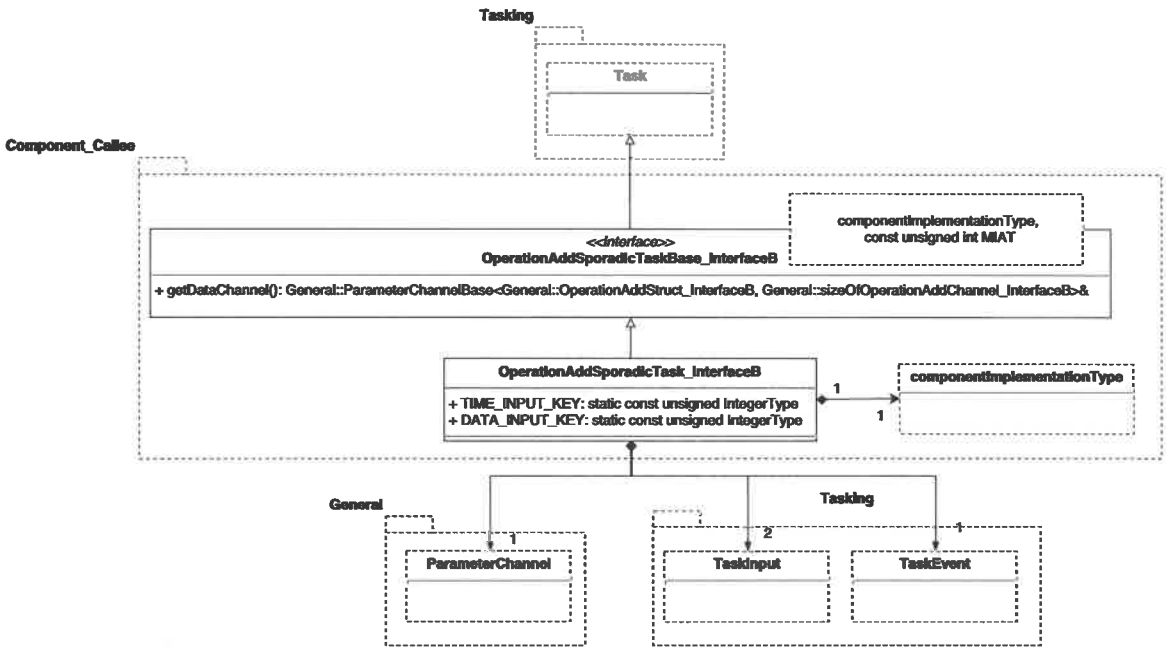

Figure 6.24.: UML class diagram representation of the task required to call `OperationAdd` in `ProvidedInterfacePort2_Component_Callee_inst` sporadically with a MIAT of 2s in the example OBSW model

- Instances of required interface ports
- Instance of component implementation
- Instances of tasks which are necessary to handle services which are called asynchronously
- Instances of tasks which are necessary to receive asynchronous events
- Instances of event emitter ports

A component container is also responsible for initializing the following:

- The instantiated component instance by providing references of the event emitter ports
- The event emitter port by providing reference of the task channel to which the emitter port needs to push the information about the event
- The provided interface ports by:
  - Triggering the operations in the provided interface ports which have the desired non-function property set as `cyclic`

  - Providing reference of the component instance in order to access the service
  - Providing references of the different task channels they need to push the data structures associated with the operations onto, in order to handle the services which have the interaction kind specifies as `asynchronous`
- The instances of tasks with reference of component instance in order to schedule the execution of the services
- The required interface ports by providing references of:
  - The instantiated component instance, in order to initialize the data structures of the operations with correct function wrappers for call-back functions
  - The respective provided interface port each one of them is bound to
- The event receiver tasks with references of the component instance and the channels which needs to be associated with their respective task inputs

For our example OBSW model: The following classes as shown in Figure 6.27 and Figure 6.28 are defined:

- `Container` in the namespace `Component_Caller` which is the container for the component instance `Component_Caller_impl_inst` and its provided and required interface slots
- `Container` in the namespace `Component_Callee` which is the container for the component instance `Component_Callee_impl_inst` and its provided interface slots

Both the containers have instances of their respective different components as explained in the general case and as shown in Figure 6.27 and Figure 6.28. They are responsible for the initialization of different components as explained in the above general description.

6.4. Code generation using Xtend

The reference implementation of the OSRA component model consists in a set of .ecore metamodels [28]. Ecore is an implementation of the Essential Meta Object Facility (EMOF), the meta-meta language by the OMG for the specification of meta-models [28]. The main advantage of using Ecore is the ready availability of graphical editors for the specification of metamodels and powerful support provided by the Eclipse Modeling Framework (EMF), which is a framework of the Eclipse development platform that permits to generate a code implementation of the metamodel entities, basic editors for the creation of models conforming to the metamodel under development [28].

It is an implementation decision in this Master thesis to use Xtend for the code generation. Xtend is a general purpose Java-like language that is completely operable with Java [2][38]. Xtend has a more concise syntax than Java and provides powerful features such as type inference, extension methods, dispatch methods, an lambda expressions and the all important multiline template expressions, which are useful when writing code generators [2][38]. Xtend also provides powerful features that make model visiting and traversing really easy, straightforward, and natural to read and maintain.

Xtext is an Eclipse framework for implementing programming languages and Domain-Specific language (DSL) [2]. Xtext helps to implement languages quickly, and most of all, it covers all the aspects of a complete language infrastructure like parser, code generator etc. Xtext uses Google Guice, which is a dependency injection framework to create and call a code generator [2]. The dependency injection pattern basically allows to inject implementation objects into a class hierarchy in a consistent way [10]. The Xtext's generator support can be used in Xtend directly to build code generators for non-Xtext based models, such as the OBSW models constructed using the OSRA component model [8].

The tutorial in [37] is used as a base in this Master thesis to construct a code generator using Xtend for non-Xtext based models and also provide a UI integration for the code generator.

6.5. Organizing the generated code

As explained in the previous sections, adopting separation of concerns even at the implementation level is one of the primary goals of this chapter and we have successfully achieved it in the discussions on software design for the infrastructural code in Section 6.3.3.

To further emphasize on separation of concerns at the implementation level, it is necessary to properly separate the generated infrastructure code into a meaningful files and a suitable file structure helping the third party software supplier to separate the automatically generated code from the code that needs to be supplied. This is of prime importance for the third party software supplier to not accidentally lose the implementations in successive code generation cycles.

The overall idea would be to:

- Generate two types of folders to clearly separate the infrastructural code entities, at the component type level from the infrastructural code entities at the component

instance level. The number of folders at the component instance level, depends on the actual number of component instances.

  - The first folder type would hold C++ classes related to its component type, required interface ports, event emitter ports, event receiver ports in the sub-folder named as `AutogeneratedCode`. It also contains C++ classes related to component implementations and because it is the unit of sub-contract, it is placed in a separate sub-folder named as `UserCode`. The third-party software supplier can alter the code in this folder safely without the fear of the code being overwritten by successive code generation cycles. Care is taken in the code generator to generate the classes in these files only once.
  - The second folder type would hold the C++ classes related to its component instances, namely, provided interface ports, component containers in the sub-folder named as `AutogeneratedCode`. As already explained the component container class for a component instance would contain provided and required interface slots, component instance itself.

- The folder named as `DatatypesInterfacesEventsAndExceptions` would hold C++ classes related to the data types, exceptions, events, parameter channel and their corresponding parameter queues.

The folder structure for our running example is listed in Appendix A.

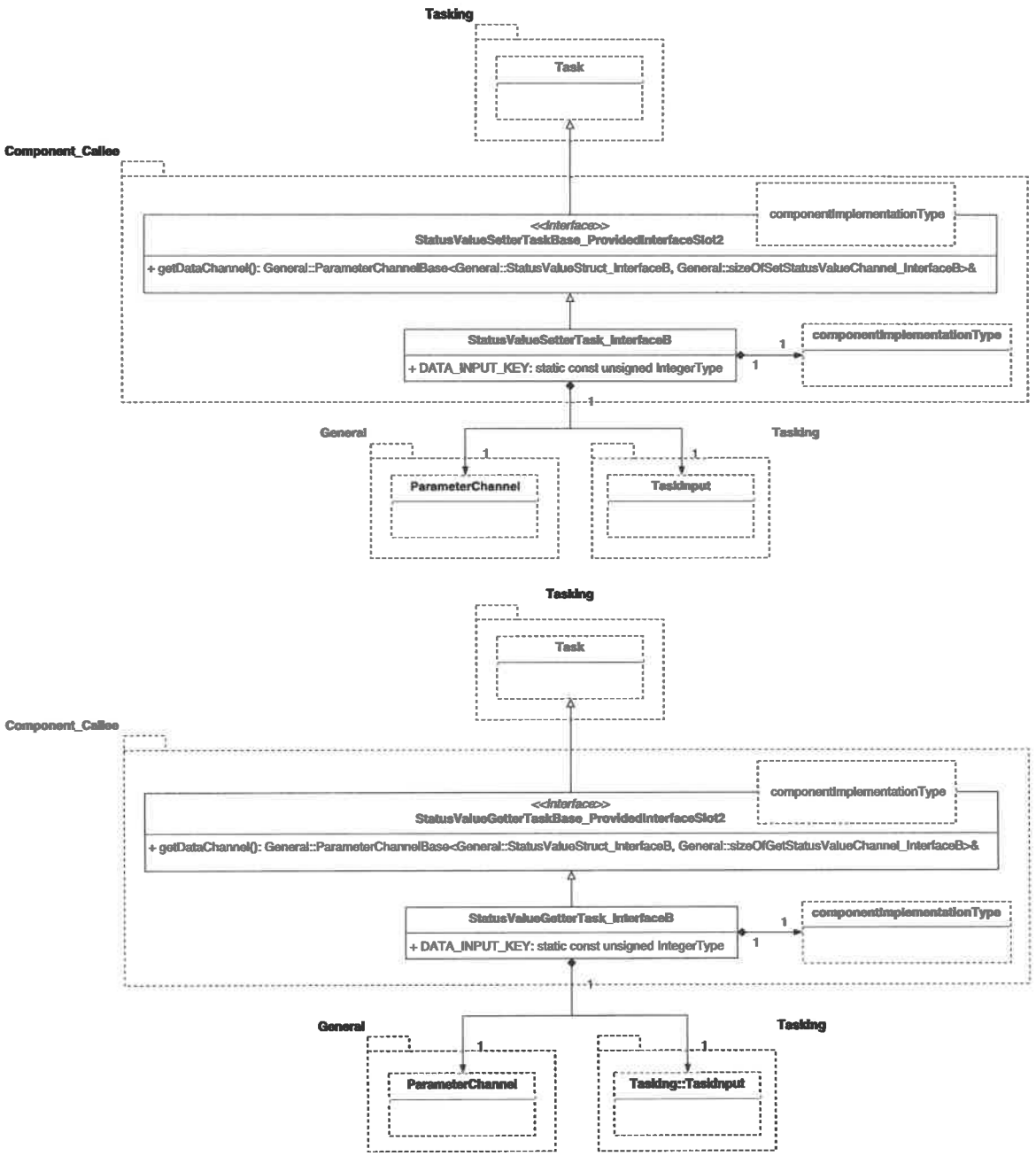

Figure 6.25.: UML class diagram representation of the tasks required to set and get the values of the interface attributes asynchronously in `ProvidedInterface Port2` in the example OBSW model

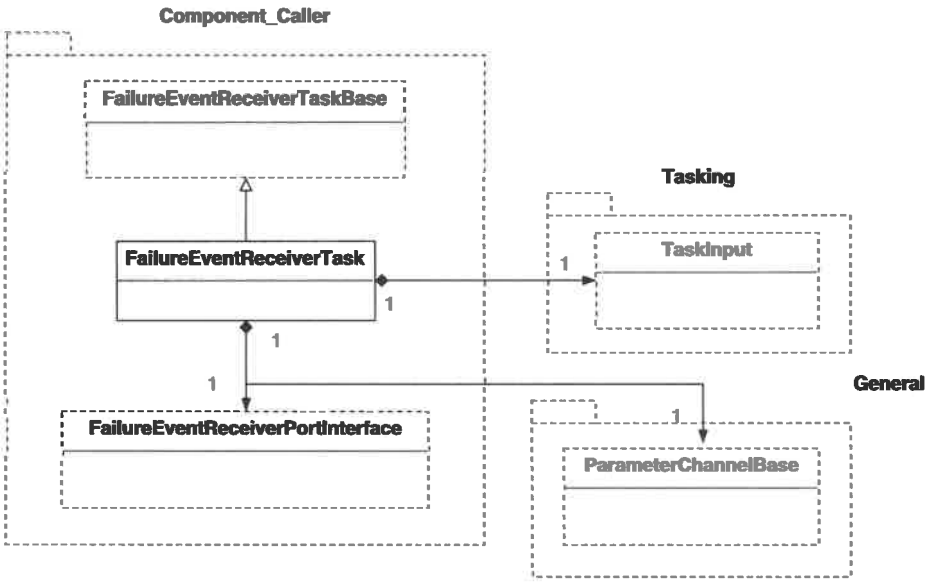

Figure 6.26.: UML class diagram representation of the task required for the reception of the `FailureEvent` in the example OBSW model

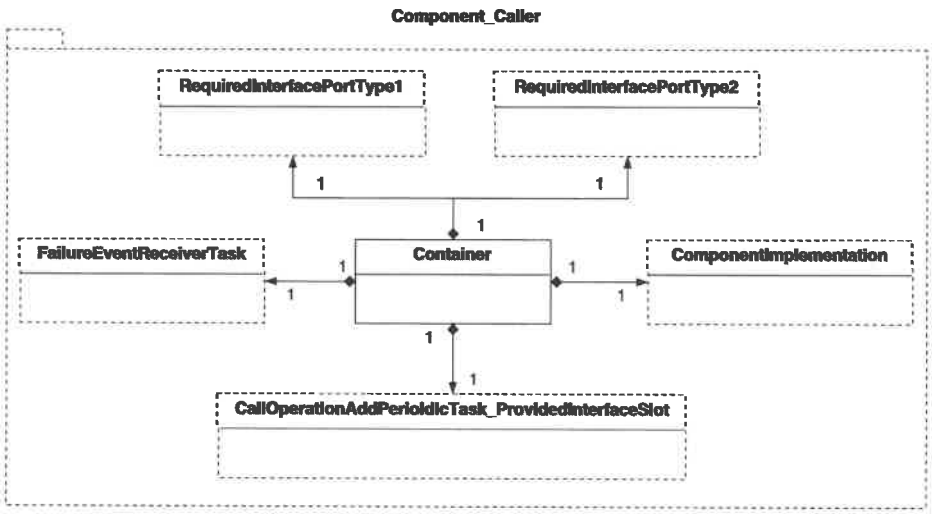

Figure 6.27.: UML class diagram representation of the container for `Component_Caller` in the example OBSW model

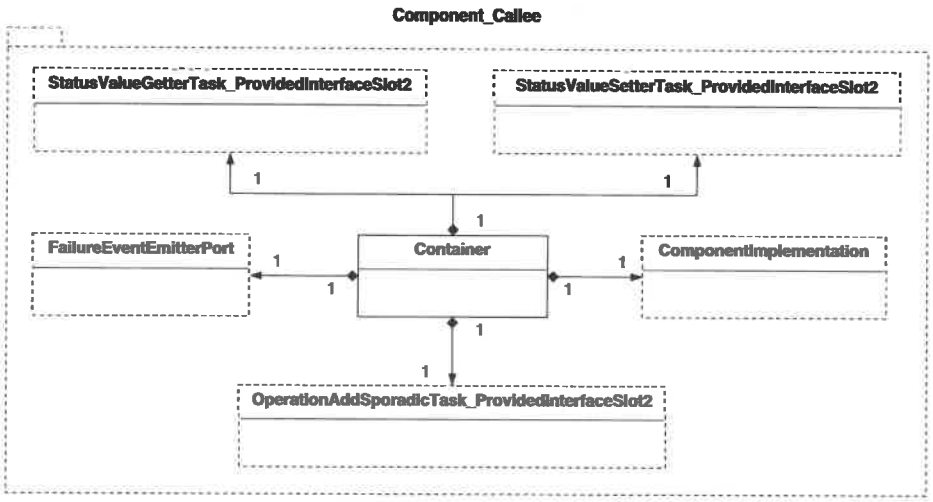

Figure 6.28.: UML class diagram representation of the container for `Component_Callee` in the example OBSW model

Chapter 7

Evaluation of the code generator

7.1. Introduction

By using model-driven engineering tools (MDE) tools for code generation, it is possible to generate software code automatically and achieve extremely high developer productivity rates of thousands of function points and millions of lines of code per person-month [16]. But, as we have seen in the previous chapters, the MDE approach consists of more than code generation tools; It defines the entire software-engineering approach that can impact the entire lifecycle from requirements gathering through sustainment [29][1].

It is important to consider these tools and methods in the context of a particular system acquisition i.e., the MDE methods and tools need to be aligned with the system acquisition strategies, which would in turn improve system quality, reduce time to field, and reduce sustainment cost [16]. System acquisition strategies include:

- Securing the necessary data rights and licensing for tools, models, generated code, run-time libraries, frameworks, and other supporting software.
- Reviewing and evaluating appropriate artifacts introduced by the MDE tools at the right time in the acquisition cycle.
- Approaches to manage program risks, include risk identification and mitigation.

If the methods and tools do not align with the system acquisition strategy, using them can result in increased risk and cost in development and sustainment[16]. The acquirers in government or large commercial enterprises have the challenge of selecting contractors to develop their systems. The tools and processes selected by the contractors and developers have direct impact on the software quality concerns of the acquirer, who often has little influence on the selection of these tools and processes. The tool acquirer would then have to answer the following acquisition evaluation questions [16]:

- Do the engineering processes and associated development tools match the desired acquisition strategy?
- Do the tools support the developer's software development methodology?
- Are the code generation tools capable of integrating with other development and management tools to support measurement and monitoring of the development progress?
- Will the selected development methodology with its associated tools be available and compatible for the expected lifecycle of the system?

This chapter subjects the code generator developed as a part of the Master thesis to evaluation methods listed in [16] and provide necessary inputs for conducting the acquisition evaluation.

7.2. Selection and evaluation methods of a MDE tool for code generation

The step-by-step MDE tool selection process defined in [16] makes use of the Plan Establish Collect Analyze (PECA) method [6] as shown in Figure 7.1.

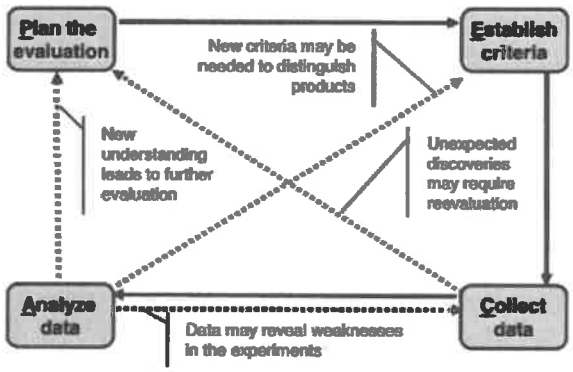

Figure 7.1.: The PECA Process

Source: [6]

As a part of the `Establish criteria` step in the PECA process, the acquirer must establish criteria with which he has to decide whether a particular tool for automatic code generation is suitable for a specific system acquisition. Such a criteria can be developed using risk taxonomy [7] which ensures that all relevant acquisitions strategies

are covered i.e., it provides a checklist to ensure all potential risks are considered [16]. The risk taxonomy has three main sections [7]:

Product engineering This covers activities that create a system that satisfies the specified requirements and customer expectations. Risks in this area generally arise from requirements that arise from requirements that are technically difficult to achieve, inadequate requirements and design analysis, or poor design and implementation quality.

Development environment This includes risks related to the development process and system, management methods, and work environment.

Program constraints This cover risks that arise from factors external to the project.

To establish a criteria for a particular program or project, it is necessary for the program to first scan the risk taxonomy and identify those areas that apply for the project. Each risk creates one or more acquisition concerns , which may refine the program risk or indicate how certain tool features or capabilities might help mitigate the risk [16].

As part of the `Collect data` step in the PECA process, a vendor self-assessment questionnaire is prepared as in [16] which is given to the MDE tool vendors, to provide data needed to make the tool selection decision.

As a part of the `Analyze data` in the PECA rocess, it is then necessary to position these acquisition concerns in the specific program context and finally decide whether acquiring a particular tool is beneficial for the project.

In this Master thesis, a subset of evaluation criteria is chosen from Appendix A in [16]. The risk areas and the acquisition concerns which are meaningful in the scope of this Master thesis are chosen. For example, the [16] discusses about potential risks in the Process Management area of the project and these are of no interest in this Master thesis and are not considered. Each of the acquisition concerns in Appendix A of [16] are then linked to the questions in the vendor self-assessment questionnaire listed in Appendix B of the [16]. For our chosen subset of potential areas and the acquisition concerns within this Master thesis, an attempt is made to answer the corresponding linked questions in the questionnaire. The tables Table 7.1, **??**, Table 7.3, Table 7.4, Table 7.5, Table 7.6 showcase these efforts.

Table 7.1.: Evaluation Criteria - Product Engineering Risk Area (Requirements)

Risk area	Potential Acquisition Concerns Related to MDE Tools for Automatic Code Generation	Answers to the linked questions in the questionnaire
Requirements		
Stability	Responding to requirements changes may necessitate operating on partially complete models and performing refactoring or rework on models.	The OBSW models designed using the OSRA Component Model should be subjected to model validation against the OSRA Specification Compliance and the SCM Metamodel Compliance before it is subjected to automatic code generation. In that case, the OBSW model, even if partially complete can be subjected to code generation if it clears the model validation step
	Communication with stakeholders is partially important to resolve requirements issues, so tool features that support this become more important	It is possible to annotate each of the model entities while constructing the OBSW model with information which can be used for communicating with the stakeholders. There is no additional documentation tool which comes with the OSRA SCM tool suite
	Interfaces between the software modeling tools and the requirements management tools promote co-evolution of requirements and software	There is no support for tracing requirements into model elements at the current state of development of OSRA SCM
Completeness Clarity Validity	In addition to the concerns noted above about requirements stability, the ability to execute or simulate the execution of the model can help validate requirements completeness	At the current stage of the development of the OSRA SCM, it is not possible visualize the execution of the model. It is only possible to create static models of the OBSW and it is not possible to trace the flow of execution through the model, or inject data or events into the model
Feasibility	The ability to perform analysis of the model for qualities such as latency, throughput and consistency can help demonstrate the feasibility of requirements	The model-based static analysis of the OBSW model is not possible at the current stage of development of the OSRA. Step 10 of the overall software development process in Section 3.2 in chapter Chapter 3 gives an idea about the analysis of latency, throughput etc. which can be performed on the OBSW model in the future
Scale	Limitation on the size or complexity of the model that can be represented, analyzed, or transformed by the tool will limit the scale of the system that can be created	There are no size and complexity limitations for representing the OBSW models using the OSRA SCM tools