Chapter 6

Infrastructural code generation

6.1. Introduction

After designing an OBSW model using the OSRA SCM model editor and following the component-based software development approach that comes with it, the OBSW model entities need to be mapped to the infrastructure code. The reference programming model for OSRA, discussed in the previous chapter helps us in progressing towards this goal. But, it is necessary to understand the overall design approach for the generated code and briefly present the abstractions that will be offered to the software supplier. This chapter, deals with these things in detail. Similar efforts from the Artemis JU CHESS project [30], provide the perfect base for discussions in this chapter of the Master thesis.

6.2. User model entities in the Platform Independent Model (PIM) phase

A detailed description of all the modeling entities that the software architect can use, can be found in the specification of the metamodel for the OSRA component model [28]. However a brief description of them is noteworthy here:

Datatypes The software architect can create a set of project-specific data types and constants using the `Data Types` language unit of the `CommonTypes` metamodel and the language unit is designed to provide the software architect an expressive power comparable to the languages with strong types (e.g. Ada).[28]. The supported type definitions are boolean types, integer types, float types, enumeration types, fixed point types, array types, structured types, string types, union types, alias

types, opaque types, external types and unconstrained types. Some of these data type definitions are obvious for readers with programming skills in typed languages such as Ada, C or C++.

Interface An interface is a specification of coherent set of services and it represents the definition of a contract. An interface is defined independently of the entities implementing it (e.g. Component type). An interface may enlist declaration of operations, which are the functional services that shall be offered by the entities implementing it. The services include a name, set of ordered parameters and one or more exceptions that they might throw when things go wrong during the handling of the service. Parameters are typed with one of the types mentioned above and have a mode (`in`, `out` or `inout`). A component type may expose one or more interfaces and the same interface can be exposed by different component types. An interface may also contain the declaration of one or more interface attributes, which are the parameters that are accessible via the interface implementations.

Component type A component type is an entity which specifies the external interfaces of a software component and are defined in isolation and are used to declare relationships with the other components and system in general. It conforms to the principle of encapsulation and as a consequence, all the interactions with other components are performed exclusively via its explicitly declared interface. A component type usually encompasses:

- A list of provided interface ports
- A list of required interface ports
- A list of dataset emitter ports
- A list of dataset receiver ports
- A list of event emitter ports
- A list of event receiver ports

Component implementation It is an entity that represents a concrete realization of a component type. It is functionally identical to the component type except that the source code is added to the component implementation and may also define number of component implementation attributes.

Component instance It is an instantiation of a component implementation and hence contains all the instantiations of the structural features, such as provided and required interface ports. It also contains instantiation of all attributes (interface attributes, component type attributes and component implementation attributes). It is also the elementary deployment unit for the OBSW model [28].

6.3. Mapping of design entities to the infrastructural code

As the generated code should target the Tasking framework, which is the target computational model in this Master thesis and because the Tasking framework is written in C++, the following sections explains the mapping of design entities to the infrastructure code that will be generated in C++.

On analyzing the specifications of the metamodel for the OSRA component model [28], it is clear that there are different corner cases that can arise during the construction of the OBSW models using OSRA component model and it is necessary that these corner cases are effectively handled in the software design for the infrastructural code. The following sections try to build an OBSW model keeping the the corners cases in mind and attempt to explain the overall design approach.

6.3.1. Corner cases arising during the construction of OBSW model using OSRA component model

The different corner cases which can arise are:

- Multiple provided interfaces which refer to the same interface type are promoted by the container of a component
- Multiple required interfaces which refer to the same interface type are subsumed by the container of a component
- Multiple interfaces provide exact same operations
- Multiple implementations per component type

The first and second corner cases are handled in the following example. But, the other cases will be treated directly in the later section, which deals with the software design for the generated infrastructure code.

6.3.2. An example OBSW model

Our simple OBSW model, yet effective to serve the intended purpose, is built as per the proposed component-based development approach explained in the section Section 3.2 in chapter Chapter 3. As already mentioned in that section, the component-based approach puts a lot of emphasis on the definition of component interfaces [29] and it is followed here as well. Components are built from scratch using newly defined interfaces.

All model entities defined here are instantiations of the modeling entities specified in the metamodel [28]. The OBSW model is designed using the OSRA SCM model editor mentioned in the section Section 3.5 in chapter Chapter 3. The model entities from the OSRA SCM model editor can be exported as images and they are used in this sub-section for illustration purposes.

In this simple example, two simple components are designed. The first component requests for a service `OperationAdd` which can add two numbers, as the name suggests and this service is implemented in the second component. Another service namely `CallOperationAdd` is implemented in the first component, which can be requested to trigger the OBSW model. Different non-functional properties, as explained later in this section, are strewn on the required and provided interfaces of these components to make example a bit more interesting and also to capture the above mentioned first and second corner cases in the example.

Step 1: Definition of data types and events As the Master thesis requires to emphasize more on effectively capturing interactions and concurrency semantics required for communication between the designed components, the data types chosen in this example are fairly simple. But it is important to note that the scheme of mapping of these simple data types to the infrastructural code (explained in the later sections), can be scaled to fairly complex data types as well. The data types, exception types and the event type used in this example are as shown in Figure 6.1.

- Two data types namely `FixedLengthStringType` and `IntegerType` of type `UNSIGNED` are defined and they are named as `StringType` and `IntegerType` respectively.
- Three exception types, named as `OperandException`, `MemoryException` and `OverflowException` are defined.
- An `Event` type, which can be used for asynchronous notifications [28] is instantiated and it is named as `FailureEvent`. Two event parameters are also instantiated as shown in Figure 6.1.

Step 2: Definition of interfaces Two interfaces namely `InterfaceA` and `InterfaceB` are designed as shown in Figure 6.1. `InterfaceA` has only one single operation by name `CallOperationAdd` and `InterfaceB` has an operation by name `OperationAdd` and an interface attribute of data type `IntegerType` and named as `m_StatusValue`.

- The operation `CallOperationAdd` is a parameterless operation and it is intended to be the service which can in turn request the service which can add two numbers.

- The operation `OperationAdd` in `InterfaceB`, as the name suggests, is intended to be the service which can add two numbers, send back the results and raise a pre-defined exception if necessary. It has three operation parameters and can throw different exceptions as mentioned in Figure 6.1..
- The interface attribute `StatusValue` in `InterfaceB` is of type `CFG` and it indicates that the interface attribute is a configurable parameter [28]. As a result, two operations for the purpose of setting and getting the values of the interface attribute are defined.

need to be generated ?!

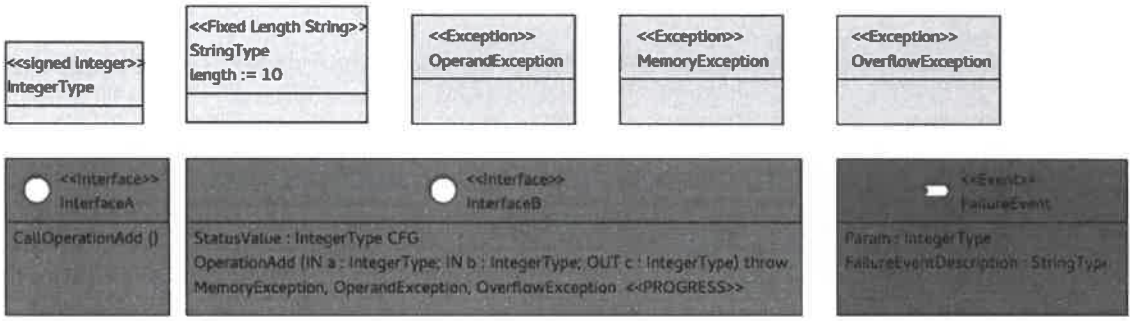

Figure 6.1.: Data types, events, exceptions and interfaces diagram

Step 3: Definition of component types Component types namely `Component_Caller` and `Component_Callee` which form the basis for a reusable software asset are defined as shown in the Figure 6.2.

`Component_Caller` has:

- Provided interface port `ProvidedInterfacePort` which refers to `InterfaceA`.
- Required interface port `RequiredInterfacePortType1` which refers to `InterfaceB`.
- Required interface port `RequiredInterfacePortType2` which refers to `InterfaceB`.
- Event receiver port `FailureEventReceiverPort` which refers to `FailureEvent`.

The desired interaction kind for the operations in the required interface ports of `Component_Caller` are as shown in the Table 6.1.

`Component_Callee` has:

- Provided interface port `ProvidedInterfacePort1` which refers to `InterfaceB`.
- Provided interface port `ProvidedInterfacePort2` which refers to `InterfaceB`.

Table 6.1.: Desired interaction kind for operations in the required interface ports

Required interface ports	Operations	Interaction kind
RequiredInterfacePortType1	OperationAdd	synchronous
	Interface attribute setter	synchronous
	Interface attribute getter	synchronous
RequiredInterfacePortType2	OperationAdd	asynchronous
	Interface attribute setter	asynchronous
	Interface attribute getter	asynchronous

- Event emitter port FailureEventEmitterPort which refers to FailureEvent

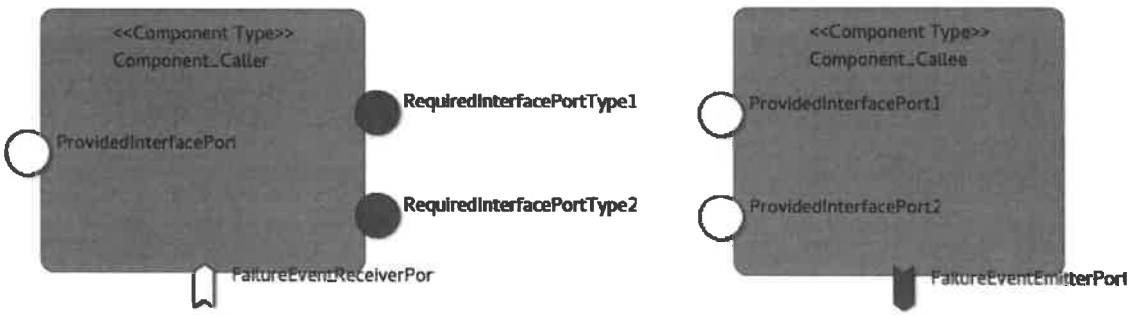

Figure 6.2.: Component types diagram

Step 4: Definition of component implementations Component implementations are created from the component types.

Component_Caller has one component implementation named as Component_Caller_impl and Component_Callee has one component implementation named as Component_Callee_impl. The component implementation Component_Callee_impl implements the means to store the attribute Param of InterfaceB, that is exposed through its provided interface ports, namely ProvidedInterfacePort1 and ProvidedInterfacePort2.

No maximum memory footprint for component implementations are defined or no detailed design activity of the component implementations are performed as they are not of concern in this Master thesis.

Step 5: Definition of component instances The component instances are the instances of component implementations [29].

Two component instances as shown in Figure 6.3 are defined, namely:

Table 6.2.: Non-functional property for the operation in the provided interface slot

Provided interface slot	Operation	Non-functional property
`ProvidedInterfaceSlot`	`CallOperationAdd`	`Cyclic`, Period = 2s

- `Component_Caller_impl_inst` which is an instantiation of `Component_Caller_impl`
- `Component_Callee_impl_inst` which is an instantiation of `Component_Callee_impl`

Step 6: Definition of component bindings Component bindings as shown in Figure 6.3 are defined:

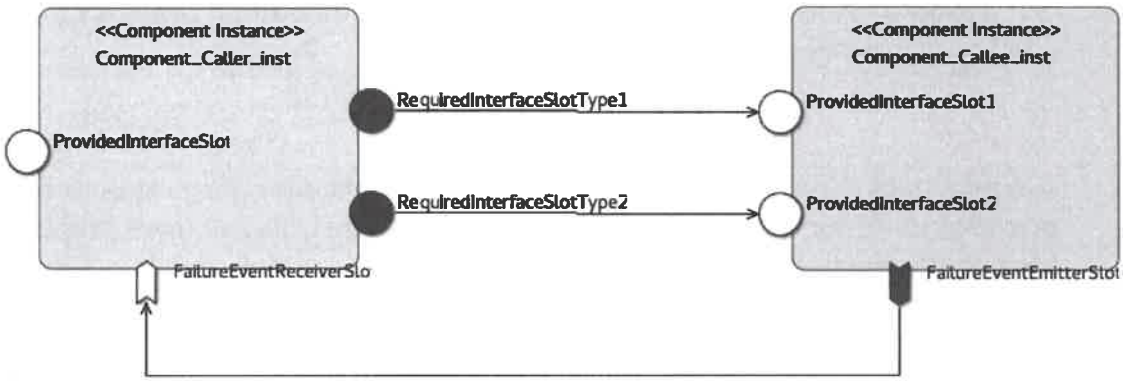

Figure 6.3.: Component instances diagram

Step 7: Specification of non-functional attributes The non-functional properties are defined on the component instances and the component bindings defined in the previous step. The non-functional properties language unit of the specification of a metamodel provides a Value Specification Language (VSL) unit, which permits the specification of the the non-functional properties qualified with a measurement unit [28]. The VSL is used here to define values of non-functional properties with a measurement unit.

For the provided interface slot in the component instance `Component_caller_impl_inst`, the following non-functional property is specified as shown in Table 6.2.

For the provided interface slots in the component instance `Component_callee_impl_inst`, the following non-functional properties are specified as shown in Table 6.3.

Table 6.3.: Non-functional properties for the operations in the provided interface slots

Provided interface slots	Operations	Non-functional properties
ProvidedInterfaceSlot1	OperationAdd	Protected
	Interface attribute setter	Protected
	Interface attribute getter	Unprotected
ProvidedInterfaceSlot2	OperationAdd	Sporadic, MIAT = 2s
	Interface attribute setter	Protected
	Interface attribute getter	Protected

Table 6.4.: Non-functional property for event reception

Event receiver slot	Event	Non-functional property
FailureEventReceiverSlot	FailureEvent	Protected

It is important to note that the WCET and deadline values for the operations in the provided interface slots are not handled, as the safeguarding of these properties are not of concern in this Master thesis.

For the event receiver slot in the component instance `Component_caller_impl_inst`, the following non-functional property is specified as shown in Table 6.4.

Step 8: Definition of the physical architecture The hardware topology provides a description of the system hardware. As hardware modeling is not of concern in this Master thesis, a simple hardware topology as shown in Figure 6.4 is considered.

A processor board with a processor and a processor core is instantiated. Two connection docks are attached to the processor board and a bus is used to connect the connection docks. The component instances are deployed on the processor core and the component bindings are deployed on the bus.

This OBSW model is subjected to model validation against the OSRA Specification Compliance and the SCM meta-model compliance, in the OSRA SCM editor [9]. Only after the OBSW model is successfully validated, can the OBSW model be considered as a suitable candidate for automatic generation of infrastructure code [9].

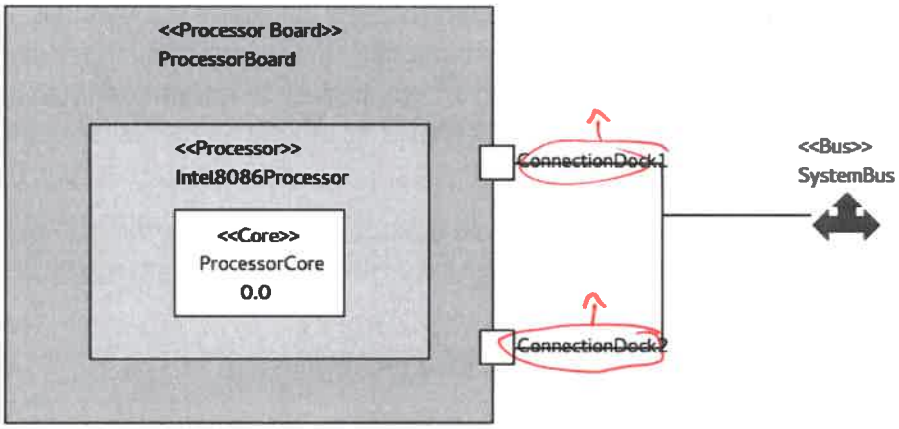

Figure 6.4.: Hardware topology diagram

6.3.3. Software design approach for the generated code

This section deals with the software design approach for the generated infrastructure code. Some of the necessary good characteristics of a software such as reusability, separation of concerns and minimal complexity are targeted already at the reference architecture level and it needs to be safeguarded at the generated software level as well. The other main characteristics of the generated software which are of utmost value are [21]:

Testability The software must be testable and there must be suitable constructs in the generated software which assist in writing automated unit tests or user driven tests to test it. In this Master thesis, it is taken care that the generated C++ classes have corresponding abstract base classes, using which mock classes can be constructed easily for the purpose of testing. More about this is explained in the next chapter which focuses on the results and further scope of this Master thesis

Extensability and Refactorability The software generated must be loosely coupled so that the entire code base is more resilient to changes and extensions. Dependency injection software design pattern [10] is used wherever appropriate to make the generated software loosely coupled

Portability The generated software must be portable across systems and environments. The data types used in the generated software should be portable across multiple platforms and the data type standardizations from C++11 are made use of in the generated software

High fan-in The generated software is designed in a way that a particular C++ class is used by large number of other C++ classes.

Low-to-medium fan-out The generated software is designed in a way that a particular C++ class uses not many classes, so that the complexity of the generated software is not too high. Unfortunately, this characteristic is not completely respected in this software design, because the complexity of a particular class depends on the complexity of the user model.

Unified Modeling Langauge (UML) class diagrams, wherever appropriate are judiciously used in this section to show a high level representation of the generated C++ classes and datastructures.

Each of the following sub-sections, is divided into two parts:

- The first part throws light on the idea of how a mapping of a given model entity to an infrastructural code entity can be done
- The second part makes the approach clear by taking the reference of our OBSW example model discussed in the previous section

The following sub-sections also include UML class diagrams wherever appropriate. The standard legends from the UML diagrams are used. The following additional legends are introduced:

- A UML class in dotted notation means that the explanation for that particular has been given in the previous sections or would be given in the following sections
- A package notation from UML is used to indicate the namespaces that the respective classes in the UML diagram can be found

6.3.3.1. Namespaces

Namespaces from C++ are used to differentiate component types, component implementations etc. of different components. The names for the namespaces are obtained from the names of the component types in the OBSW model.

For our example OBSW model: Three namespaces, namely `General`, `Component_Caller` and `Component_Callee` are created as shown in Figure 6.5

Figure 6.5.: UML class diagram representation for different namespaces in the example OBSW model

6.3.3.2. Data types and events

A data type from the OBSW model is translated into a simple `typedef` statement from C++. Rationale ?

For our example OBSW model:

- The data type `IntegerType`, is translated to `typedef int8_t IntegerType`
- The data type `StringType`, is translated to `typedef std::string StringType`

A subset of all possible data types from the OSRA Component Model can be translated to simple `typedef` statements as shown above. More information about the subset of data types for which this successfully works is given in the next chapter.

The exception types from the OBSW models are translated into simple enumeration literals from C++. These exceptions, which can be thrown by a particular operation are grouped under an enumeration. This enumeration is further instantiated in a C++ struct. Rationale ? Alternative ?

For our example OBSW model: The three exceptions `OperandException`, `MemoryException` and `OverflowException` are translated to enumeration literals. These exceptions can be thrown by `OperationAdd`, which is defined in `InterfaceB`. Hence the enumeration literals, corresponding to the exceptions, are stored together as an enumeration named `OperationAddException_InterfaceB` as shown in the Figure 6.6. This exception is further instantiated in a C++ struct `OperationAddReport_InterfaceB` as shown in the Figure 6.6.

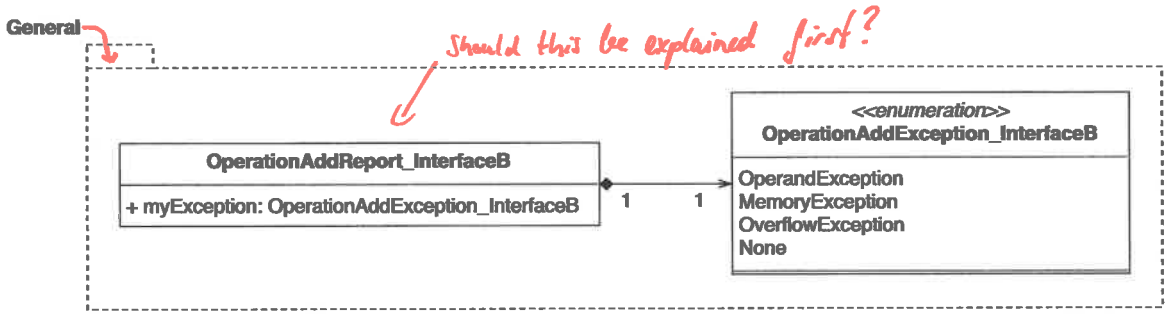

Figure 6.6.: UML class diagram representation for exceptions in the example OBSW model

An event from the OBSW model is mapped to an abstract base class and a corresponding concrete implementation class. As already explained, the abstract base classes help in improving the testability of the generated software. Appropriate setters and getters for the event parameters are declared as pure virtual methods in the abstract base class for the event and they are implemented in their corresponding concrete implementation.

For our example OBSW model: The `FailureEvent` is mapped as an abstract base class named `FailureEventInterface` and concrete implementation class named `FailureEvent`. Appropriate setters and getters for the event parameters `Param` and `FailureEventDescription` are declared as pure virtual methods in the `FailureEvent Interface` abstract base class and implemented in the `FailureEvent` concrete implementation class. An UML class diagram representation of the generated classes are as shown in Figure 6.7.

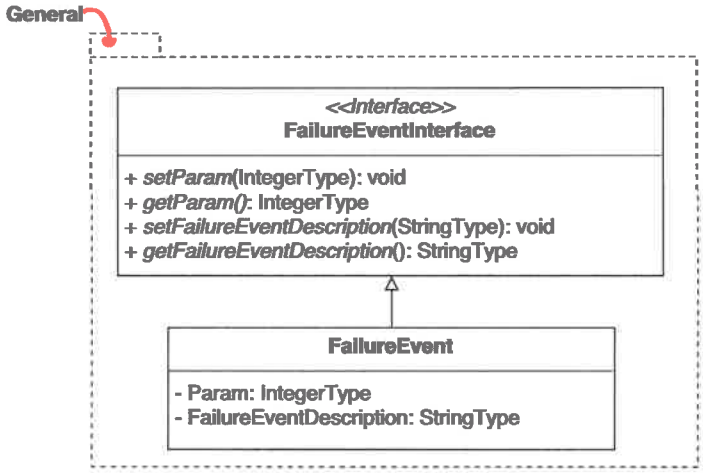

Figure 6.7.: UML class diagram representation for event in the example OBSW model

All the infrastructure code entities mentioned above are present in the namespace `General`.

6.3.3.3. Interfaces

An interface can be mapped to a C++ abstract base class, as in [30]. Constituents of this abstract base class are:

- For each interface operation, a pure virtual method is declared in the interface class. The names and data types of the input parameters for this pure virtual method corresponds to the names and data types of the interface operation parameters. The operation parameters, with `ParameterDirectionKind` as `in` are translated to constant references and the operation parameters with `ParameterDirectionKind` as `out` or `inout` are translated to plain references.
- For each interface attribute parameter of type `CFG`:
  - A class variable of name and data type corresponding to the name and data type of interface attribute is added

  - Pure virtual setter and getter methods for the interface attribute are declared. The data types and names of the input parameters in the setter and getter methods mimic the name and data type of the interface attribute.

- For each interface attribute parameter of type `MIS`, which is fixed and is not variable [28]:
  - A `const` class variable of name and data type corresponding to the name and data type of interface attribute is added.
  - No getter and setter methods are added.
- For each interface attribute of type `DAT`, which are modifiable by the component only and not by external entities [28]:
  - A class variable of name and data type corresponding to the name and data type of interface attribute is added.
  - No getter and setter methods are added.

For our example OBSW model The C++ classes shown in Figure 6.8 are generated:

- `InterfaceA` along with the operation `CallOperationAdd` is mapped to an abstract base class `InterfaceA` with a pure virtual method `CallOperationAdd`.
- `InterfaceB` has one operation `OperationAdd` and one interface attribute parameter `StatusValue` of type `CFG`. These are mapped to an abstract base class named `InterfaceB` with the following pure virtual methods:
  - `OperationAdd` with two input parameters of type `const IntegerType&` and one input parameter of type `IntegerType&`.
  - getter method for the interface attribute `StatusValue` with an input parameter of type `IntegerType&`.
  - setter method for the interface attribute `StatusValue` with an input parameter of type `const IntegerType&`.

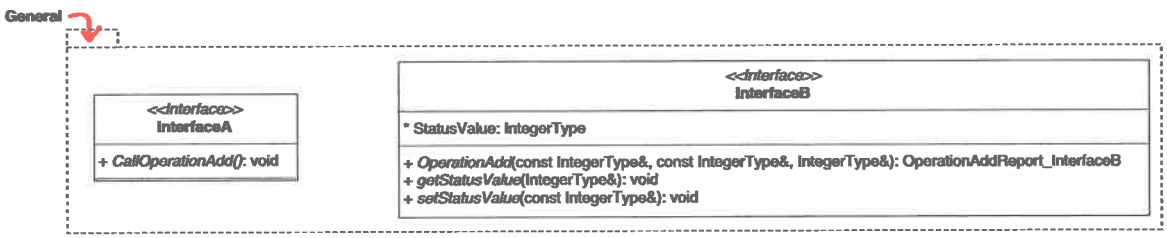

Figure 6.8.: UML class diagram representation for interfaces in the example OBSW model

Listing 6.1 Code excerpt from the generated code for `InterfaceA_Helper`

```
class InterfaceA {
public:
      InterfaceA(){}
      virtual ~InterfaceA(){}
      virtual void callOperationAdd(void) = 0;
};

class InterfaceA_Helper: public InterfaceA {
public:
      InterfaceA_Helper(){}
      virtual ~InterfaceA_Helper(){}
      virtual void callOperationAdd_InterfaceA(void) = 0; //New pure virtual method
      virtual void callOperationAdd(void)final {return (callOperationAdd_InterfaceA());}
};
```

Because of the corner case that multiple interfaces can have exactly same operations, it is necessary to refine these interfaces using the interface helper abstract base classes as shown in Figure 6.9. In each interface helper class:

- Implementations for all the inherited pure virtual methods from the parent interface are provided.
- The implementations consist of simple method calls to new pure virtual methods.
- These new pure virtual methods have method signatures same as the pure virtual methods that are inherited and implemented. However, it is important to note that the names of these new pure virtual methods are different from the inherited pure virtual methods, as shown in the Listing 6.1.

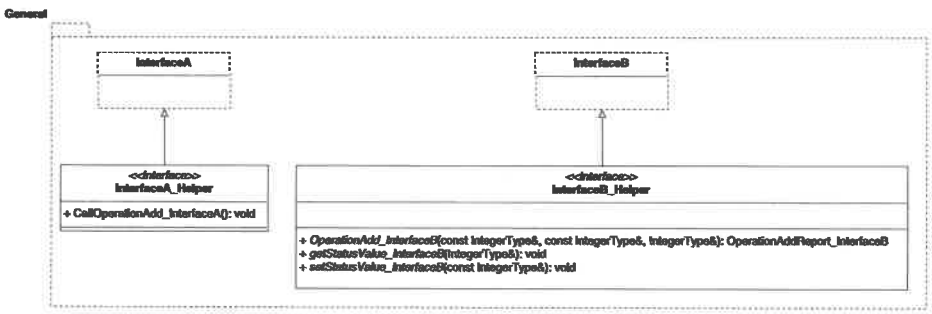

Figure 6.9.: UML class diagram representation for interface helpers in the example OBSW model

For our example OBSW model:

- `InterfaceA_Helper` is defined, which inherits from the interface `InterfaceA` and which implements the pure virtual method in the parent interface `InterfaceA`. The implementation contains a simple call to a new pure virtual method added to the original method name from the parent interface as shown in the Listing 6.1.
- `InterfaceB_Helper` is defined, which inherits from the interface `InterfaceB` and which implements all the pure virtual methods in the parent interface `InterfaceB`. Each implementation contain a simple call to the new pure virtual methods added. The `InterfaceB_Helper` class is designed and implemented the same way as `InterfaceA_Helper` is designed in the code excerpt in Listing 6.1

The combined effect is that now, more than one original parent interfaces (resembling model entities) can have same operations and interface attributes. The refined interfaces redefine the methods from the original parent interfaces, so that there are no confusions between operations from different interfaces. Of course, a straight forward solution would have been to incorporate the concept of namespaces from C++, but it is not suitable for this design and the reason is explained later in this section.

For each interface operation and interface attribute in an interface, a C++ struct is defined to carry around the values of the operation parameters or the values of the interface attributes. These data structures come in handy, when the interface operations or interface attribute access operations need to be accessed asynchronously. The data structures also hold general purpose polymorphic function wrappers from C++11 standard to store the call-back functions wherever appropriate.

For our example OBSW model:

- A struct `OperationAddStruct_InterfaceB` is defined as shown in Figure 6.10
- A struct `StatusValueStruct_InterfaceB` is defined as shown in Figure 6.10

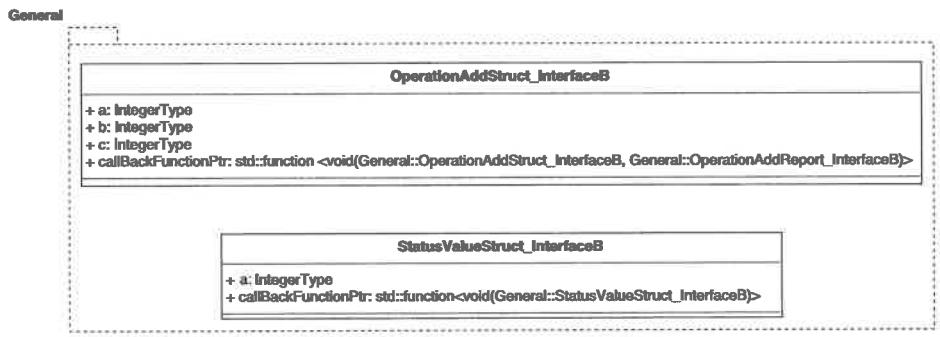

Figure 6.10.: UML class diagram representation of data structures for interface operation and interface attribute in the example OBSW model

All the infrastructure code entities mentioned above are present in the namespace `General`.

6.3.3.4. Parameter channels and parameter queues

The data structures which are defined to carry around the values of the operation parameters or values of the interface attributes, need to be pushed onto parameter channels, each one of which is supported in the back end by corresponding parameter queue.

For our example OBSW model: `ParameterChannel` and `ParameterQueue` C++ classes as shown in Figure 6.11 are defined. These classes can be reused for any OBSW model.

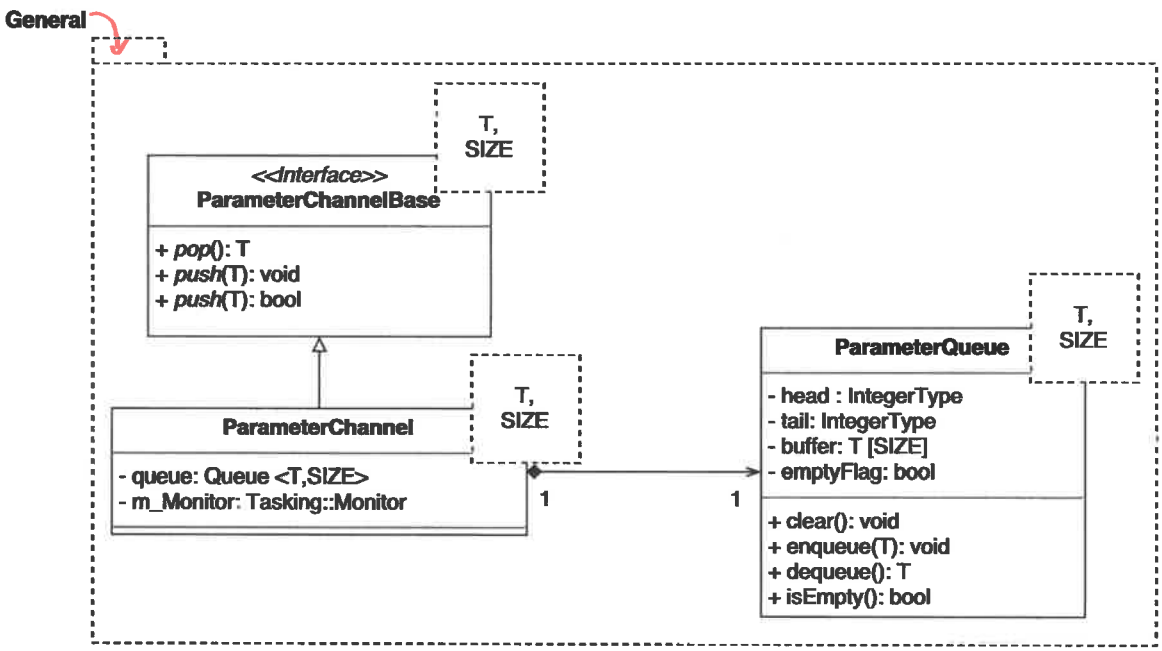

Figure 6.11.: UML class diagram representation of parameter channel and parameter queue in the example OBSW model

All the infrastructure code entities mentioned above are present in the namespace `General`.

6.3.3.5. Event emitter ports and event receiver ports

The event emitter port for a particular event is mapped as an abstract base class and a corresponding concrete implementation class in C++. The event receiver port for a

particular event is mapped only as an abstract base class. The abstract base class for event receiver port also contains pure virtual methods in order to safely interleave the reception of events.

For our example OBSW model: The `FailureEventEmitterPort` is mapped as a pair of abstract base class and a concrete implementation class as shown in Figure 6.12.

For our example OBSW model: The `FailureEventReceiverPort` is mapped to an abstract base class as shown in Figure 6.13.

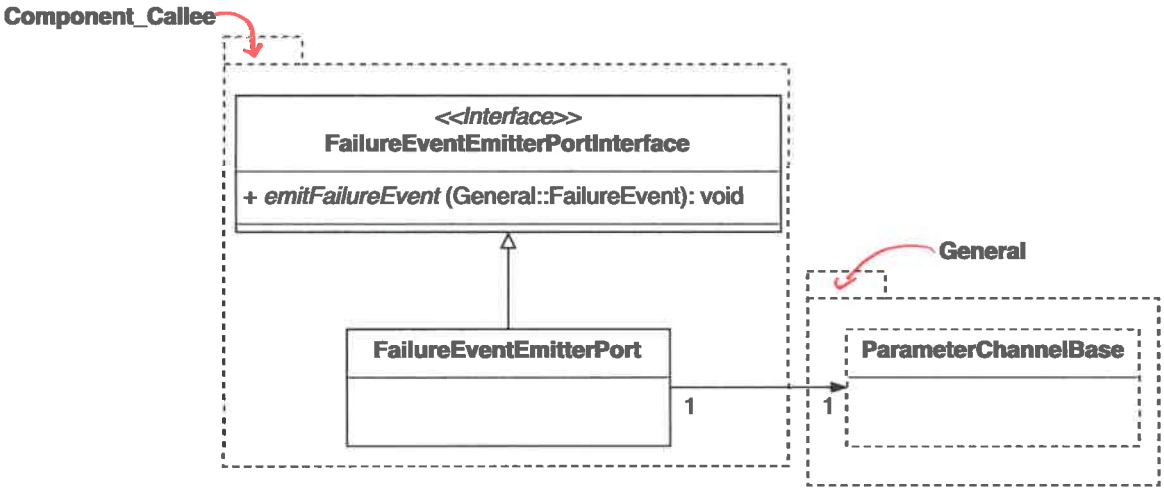

Figure 6.12.: UML class diagram representation of event emitter port in the example OBSW model

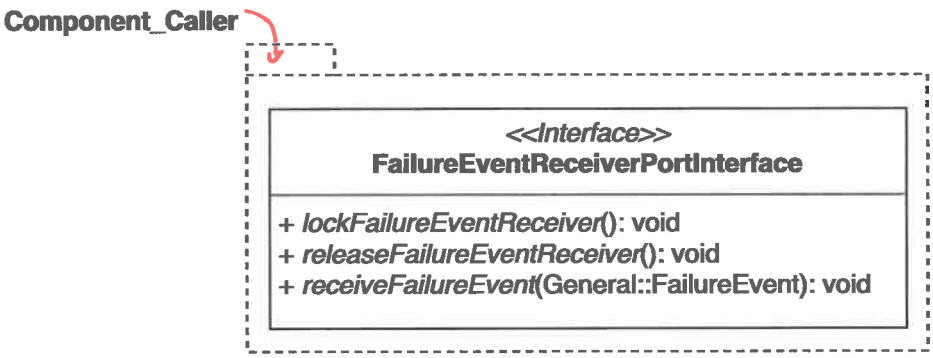

Figure 6.13.: UML class diagram representation of event receiver port in the example OBSW model

The `FailureEventEmitterPortInterface` and `FailureEventEmitterPort` classes are defined in the namespace `Component_Callee`.

The FailureEventReceiverPortInterface is defined in the namespace Component_Caller.

→ Explanation of Semantics? Idea?

6.3.3.6. Component types

A component type can be mapped to an abstract base class in C++. A component type must provide all the operations that are listed in the provided interfaces of the component [29]. Hence it inherits from all the interface helper classes which are referenced by its provided interfaces as in [30]

This is where interface helper classes, with redefined operations come in handy, because C++ does not distinguish between operations with same signatures, although they are inherited from different namespaces. A component type must also inherit from the mapped abstract base classes for event receiver ports.

A component type must also have pure virtual methods which obtain and release the semaphores for the concurrent access of different operations that it provides. In addition to these, pure virtual methods need to be added, which act as call-back functions for the operations, that the component type's required interface ports request to be released asynchronously.

For our example OBSW model:

- As shown in Figure 6.14 ComponentType in the namespace Component_Caller inherits from the InterfaceA_Helper and also inherits from the abstract base class FailureEventReceiverPortInterface. It has pure virtual methods meant for:
  - Obtaining and releasing of semaphores for concurrent accesses of the operation CallOperationAdd_InterfaceA
  - Call-back function for the operation OperationAdd
  - Call-back function for the getter operation of the interface attribute StatusValue
- A shown in Figure 6.15 ComponentType in the namespace Component_Callee inherits from the InterfaceB_Helper. It has pure virtual methods meant for:
  - Obtaining and releasing of semaphores for concurrent access of the operation OperationAdd_InterfaceB
  - Obtaining and releasing of semaphores for concurrent access of the setter and getter operations for the interface attribute StatusValue