MTConnect® Standard
Part 5.0 – Interface Interaction Model
Version 2.5.0

Prepared for: MTConnect Institute
Prepared from: `MTConnectSysMLModel.xml`
Prepared on: January 5, 2026

January 5, 2026

MTConnect Specification and Materials

The normative XMI is located at the following URL: `MTConnectSysMLModel.xml`

Table of Contents

Table of Figures

List of Tables

1 Purpose of This Document

This document, *MTConnect Standard: Part 5.0 - Interface Interaction Model* of the MTConnect Standard, defines a structured data model used to organize information required to coordinate inter-operations between pieces of equipment.

This data model is based on an *interaction model* that defines the exchange of information between pieces of equipment and is organized in the MTConnect Standard by `Interfaces`.

`Interfaces` is modeled as an extension to the *Device Information Model* and *Observation Information Model*. `Interfaces` leverages similar rules and terminology as those used to describe a component in the *Device Information Model*. `Interfaces` also uses similar methods for reporting data to those used in the *MTConnectStreams Response Document*.

As defined in *MTConnect Standard: Part 2.0 - Device Information Model*, `Interfaces` *organizes* the `Interface` types (see Figure 3). Each individual `Interface` contains data associated with the corresponding *interface*.

> Note: See *MTConnect Standard: Part 2.0 - Device Information Model* and *MTConnect Standard: Part 3.0 - Observation Information Model* of the MTConnect Standard for information on how `Interfaces` is structured in the *response documents* which are returned from an *agent* in response to a *probe request*, *sample request*, or *current request*.

2 Terminology and Conventions

Refer to *MTConnect Standard Part 1.0 - Fundamentals* for a dictionary of terms, reserved language, and document conventions used in the MTConnect Standard.

2.1 MTConnect References

[MTConnect Part 1.0] *MTConnect Standard Part 1.0 - Fundamentals*. Version 2.0.

[MTConnect Part 2.0] *MTConnect Standard: Part 2.0 - Device Information Model*. Version 2.0.

[MTConnect Part 3.0] *MTConnect Standard: Part 3.0 - Observation Information Model*. Version 2.0.

[MTConnect Part 5.0] *MTConnect Standard: Part 5.0 - Interface Interaction Model*. Version 2.0.

3 Interface Interaction Model

In many manufacturing processes, multiple pieces of equipment must work together to perform a task. The traditional method for coordinating the activities between individual pieces of equipment is to connect them using a series of wires to communicate equipment states and demands for action. These interactions use simple binary ON/OFF signals to accomplished their intention.

In the MTConnect Standard, *interfaces* provides a means to replace this traditional method for interconnecting pieces of equipment with a structured *interaction model* that provides a rich set of information used to coordinate the actions between pieces of equipment. Implementers may utilize the information provided by this data model to (1) realize the interaction between pieces of equipment and (2) to extend the functionality of the equipment to improve the overall performance of the manufacturing process.

The *interaction model* used to implement *interfaces* provides a lightweight and efficient protocol, simplifies failure recovery scenarios, and defines a structure for implementing a Plug-And-Play relationship between pieces of equipment. By standardizing the information exchange using this higher-level semantic information model, an implementer may more readily replace a piece of equipment in a manufacturing system with any other piece of equipment capable of providing similar *interaction model* functions.

Two primary functions are required to implement the *interaction model* for an *interfaces* and manage the flow of information between pieces of equipment. Each piece of equipment needs to have the following:

- An *agent* which provides:
- The data required to implement the *interaction model*.
- Any other data from a piece of equipment needed to implement the *interface* – operating states of the equipment, position information, execution modes, process information, etc.
- A client software application that enables the piece of equipment to acquire and interpret information from another piece of equipment.

3.1 Interfaces Architecture

MTConnect Standard is based on a communications method that provides no direct way for one piece of equipment to change the state of or cause an action to occur in another piece of equipment. The *interaction model* used to implement *interfaces* is based on a *publish and subscribe* type of communications as described in *MTConnect Standard Part 1.0 - Fundamentals* and utilizes a *request* and *response* information exchange mechanism. For *interfaces*, pieces of equipment must perform both the publish (*agent*) and subscribe (client) functions.

> Note: The current definition of *interfaces* addresses the interaction between two pieces of equipment. Future releases of the MTConnect Standard may address the interaction between multiple (more than two) pieces of equipment.

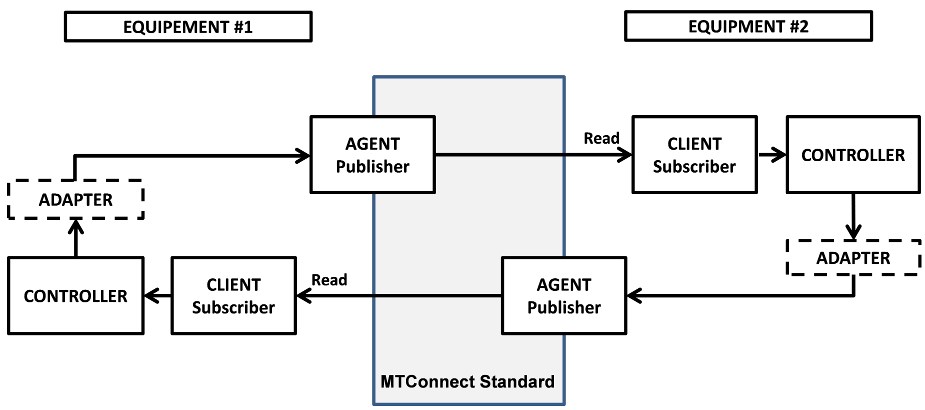

Figure 1: Data Flow Architecture for Interfaces

> Note: The data flow architecture illustrated in Figure 1 was historically referred to in the MTConnect Standard as a read-read concept.

In the implementation of the *interaction model* for *interfaces*, two pieces of equipment can exchange information in the following manner. One piece of equipment indicates a *request* for service by publishing a type of *request* using a data item provided through an *agent* as defined in *Section 4.3 - DataItem Types for Interface*. The client associated with the second piece of equipment, which is subscribing to data from the first machine, detects and interprets that *request*. If the second machine chooses to take any action to fulfill this *request*, it can indicate its acceptance by publishing a *response* using a data item provided through its *agent*. The client on the first piece of equipment continues to monitor information from the second piece of equipment until it detects an indication that the *response* to the *request* has been completed or has failed.

An example of this type of interaction between pieces of equipment can be represented by a machine tool that wants the material to be loaded by a robot. In this example, the machine tool is the *requester*, and the robot is the *responder*. On the other hand, if the robot wants the machine tool to open a door, the robot becomes the *requester* and the machine tool the *responder*.

3.2 Request and Response Information Exchange

The `DataItem` elements defined by the *interaction model* each have a `REQUEST` and `RESPONSE` subtype. These subtypes identify if the data item represents a *request* or a *response*. Using these data items, a piece of equipment changes the state of its *request* or *response* to indicate information that can be read by the other piece of equipment. To aid in understanding how the *interaction model* functions, one can view this *interaction model* as a simple state machine.

The interaction between two pieces of equipment can be described as follows. When the *requester* wants an activity to be performed, it transitions its *request* state from a `READY` state to an `ACTIVE` state. In turn, when the client on the *responder* reads this information and interprets the *request*, the *responder* announces that it is performing the requested task by changing its response state to `ACTIVE`. When the action is finished, the *responder* changes its response state to `COMPLETE`. This pattern of *request* and *response* provides the basis for the coordination of actions between pieces of equipment. These actions are implemented using `EVENT` category data items. (See *Section 4.3 - DataItem Types for Interface* for details on the `Event` type data items defined for *interfaces*.)

> Note: The implementation details of how the *responder* piece of equipment reacts to the *request* and then completes the requested task are up to the implementer.

The initial condition of both the *request* and *response* states on both pieces of equipment is `READY`. The dotted lines indicate the on-going communications that occur to monitor the progress of the interactions between the pieces of equipment.

The interaction between the pieces of equipment as illustrated in Figure 2 progresses through the sequence listed below.

- The *request* transitions from `READY` to `ACTIVE` signaling that a service is needed.
- The *response* detects the transition of the *request*.
- The *response* transitions from `READY` to `ACTIVE` indicating that it is performing the action.
- Once the action has been performed, the *response* transitions to `COMPLETE`.
- The *request* detects the action is `COMPLETE`.
- The *request* transitions back to `READY` acknowledging that the service has been performed.

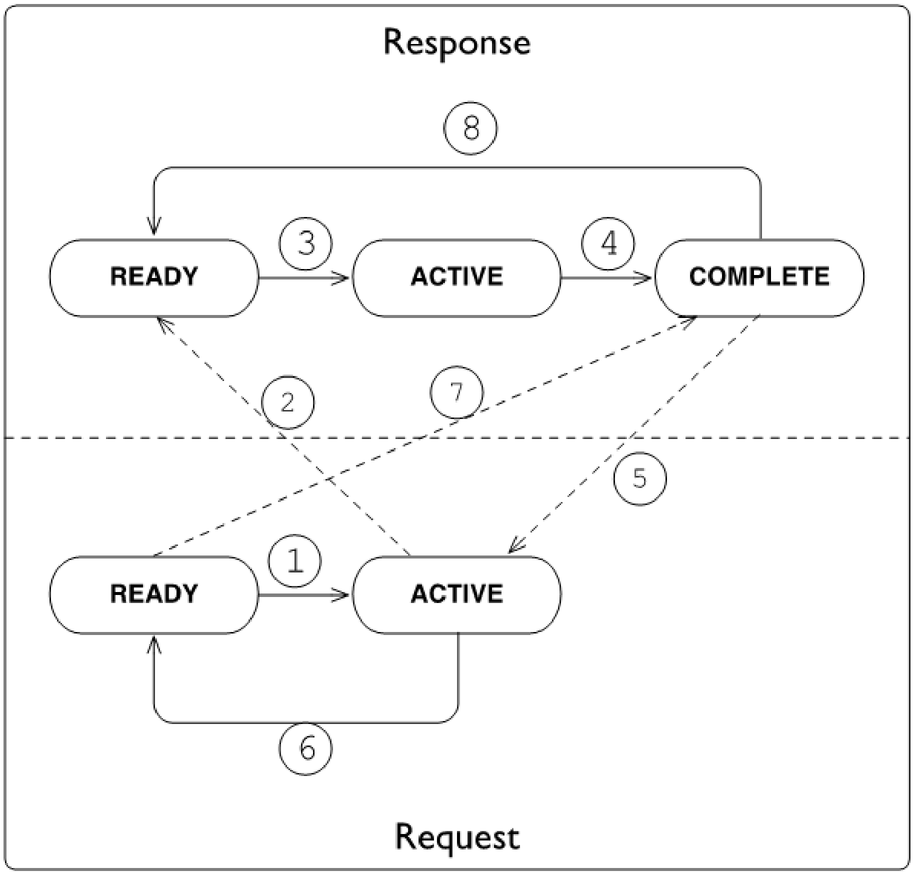

Figure 2: Request and Response Overview

- The *response* detects the *request* has returned to `READY`.
- In recognition of this acknowledgement, the *response* transitions back to `READY`.

After the final action has been completed, both pieces of equipment are back in the `READY` state indicating that they are able to perform another action.

3.3 Interface

abstract `Component` that coordinates actions and activities between pieces of equipment.

3.3.1 Commonly Observed DataItem Types for Interface

Table 1 lists the Commonly Observed DataItem Types for `Interface`.

Commonly Observed DataItem Types	Multiplicity
`InterfaceState`	1

Table 1: Commonly Observed DataItem Types for Interface

4 Interfaces for Device and Observation Information Models

The *interaction model* for implementing *interfaces* is defined in the MTConnect Standard as an extension to the *Device Information Model* and *Observation Information Model*.

A piece of equipment **MAY** support multiple different *interfaces*. Each piece of equipment supporting *interfaces* **MUST** model the information associated with each *interface* as an `Interface` component. `Interface` is an abstract `Component` and is realized by `Interface` component types.

The Figure 3 illustrates where an `Interface` is modeled in the *Device Information Model* for a piece of equipment.

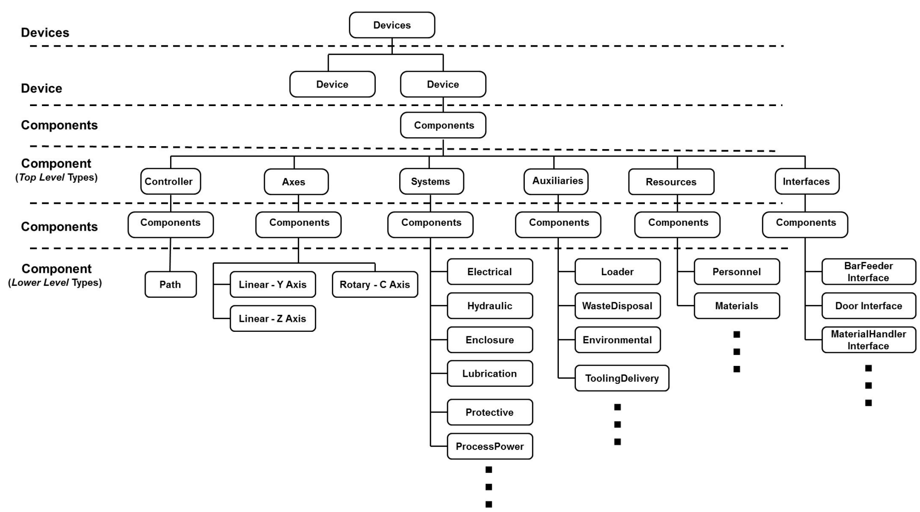

Figure 3: Interfaces in Entity Hierarchy

4.1 Interface Types

The abstract `Interface` is realized by the following types listed in this section.

In order to implement the *interaction model* for *interfaces*, each piece of equipment associated with an *interface* **MUST** provide the corresponding `Interface` type. A piece of equipment **MAY** support any number of unique *interfaces*.

4.1.1 BarFeederInterface

`Interface` that coordinates the operations between a bar feeder and another piece of equipment.

Bar feeder is a piece of equipment that pushes bar stock (i.e., long pieces of material of various shapes) into an associated piece of equipment – most typically a lathe or turning center.

4.1.2 ChuckInterface

`Interface` that coordinates the operations between two pieces of equipment, one of which controls the operation of a chuck.

The piece of equipment that is controlling the chuck **MUST** provide the data item `ChuckState` as part of the set of information provided.

4.1.3 DoorInterface

`Interface` that coordinates the operations between two pieces of equipment, one of which controls the operation of a door.

The piece of equipment that is controlling the door **MUST** provide data item `DoorState` as part of the set of information provided.

4.1.4 MaterialHandlerInterface

`Interface` that coordinates the operations between a piece of equipment and another associated piece of equipment used to automatically handle various types of materials or services associated with the original piece of equipment.

A material handler is a piece of equipment capable of providing any one, or more, of a variety of support services for another piece of equipment or a process like:

- Loading/unloading material or tooling
- Part inspection

- Testing
- Cleaning

A robot is a common example of a material handler.

4.2 Data for Interface

Each *interface* **MUST** provide the data associated with the specific *interface* to implement the *interaction model* and any additional data that may be needed by another piece of equipment to understand the operating states and conditions of the first piece of equipment as it applies to the *interface*.

Details on data items specific to the *interaction model* for each type of *interface* are provided in *Section 4.3 - DataItem Types for Interface*.

An implementer may choose any other data available from a piece of equipment to describe the operating states and other information needed to support an *interface*.

4.2.1 References for Interface

Some of the data items needed to support a specific *interface* may already be defined elsewhere in the *MTConnectDevices Response Document* for a piece of equipment. However, the implementer may not be able to directly associate this data with the *interface* since the MTConnect Standard does not permit multiple occurrences of a piece of data to be configured in an *MTConnectDevices Response Document*. `References` provides a mechanism for associating information defined elsewhere in the *information model* for a piece of equipment with a specific *interface*.

`References` *organizes* `Reference` elements.

`Reference` is a pointer to information that is associated with another entity defined elsewhere for a piece of equipment.

`References` is an economical syntax for providing interface specific information without directly duplicating the occurrence of the data. It provides a mechanism to include all necessary information required for interaction and deterministic information flow between pieces of equipment.

For more information on the `References` model, see *MTConnect Standard: Part 2.0 - Device Information Model*.

4.3 DataItem Types for Interface

Each `Interface` contains data items which are used to communicate information required to execute the *interface*. When these data items are read by another piece of equipment, that piece of equipment can then determine the actions that it may take based upon that data.

`InterfaceState` is a data item specifically defined for *interfaces*. It defines the operational state of the *interface*. This is an indicator identifying whether the *interface* is functioning or not. See *Section 4.3.4 - InterfaceState* for complete semantic details.

Some data items **MAY** be directly associated with the `Interface` element and others will be organized by a `References` element. It is up to an implementer to determine which additional data items are required for a particular *interface*.

4.3.1 Specific Data Items for the Interaction Model for Interface

A special set of data items have been defined to be used in conjunction with `Interface`. They provide information from a piece of equipment to *request* a service to be performed by another associated piece of equipment; and for the associated piece of equipment to indicate its progress in performing its *response* to the *request* for service. .

Many of the data items describing the services associated with an *interface* are paired to describe two distinct actions – one to *request* an action to be performed and a second to reverse the action or to return to an original state. For example, a `DoorInterface` will have two actions `OpenDoor` and `CloseDoor`. An example of an implementation of this would be a robot that indicates to a machine that it would like to have a door opened so that the robot could extract a part from the machine and then asks the machine to close that door once the part has been removed.

When these data items are used to describe a service associated with an *interface*, they **MUST** have one of the following two `subType` elements: `REQUEST` or `RESPONSE`. These **MUST** be specified to define whether the piece of equipment is functioning as the *requester* or *responder* for the service to be performed. The *requester* **MUST** specify the `REQUEST` `subType` for the data item and the *responder* **MUST** specify a corresponding `RESPONSE` `subType` for the data item to enable the coordination between the two pieces of equipment.

These data items and their associated `subType` provide the basic structure for implementing the *interaction model* for an *interface* and are defined in the following sections.

Figure 4 and Figure 5 show possible state transitions for a *request* and *response* respectively. The state machine diagrams provide the permissible values of the observations for the `DataItem` types listed in this section.

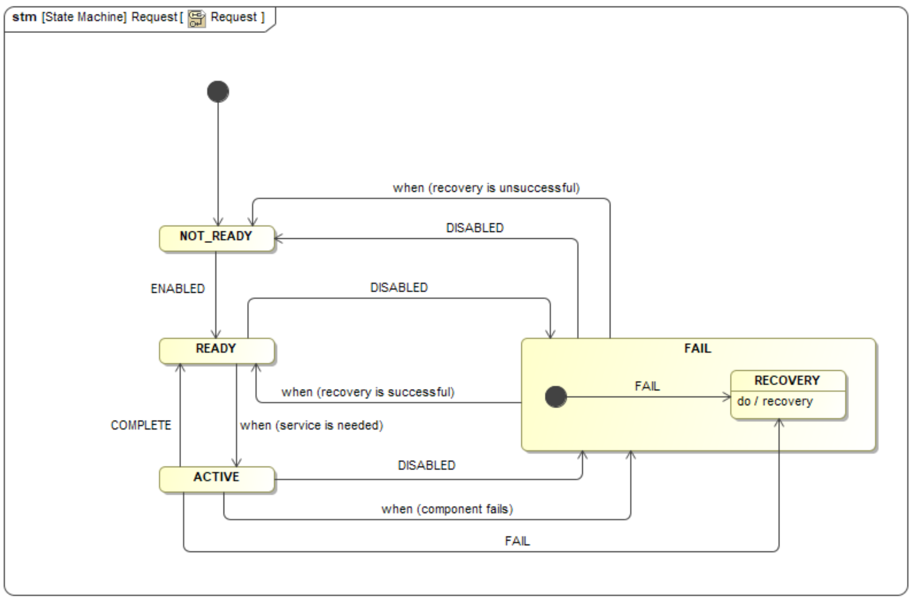

Figure 4: Request State Machine

4.3.2 CloseChuck

A `subType` **MUST** always be specified.

4.3.2.1 Subtypes of CloseChuck

- `REQUEST`

  operating state of the *request* to close a chuck.

  `RequestStateEnum` Enumeration:

  - `ACTIVE`

    requester has initiated a *request* for a service and the service has not yet been completed by the *responder*.

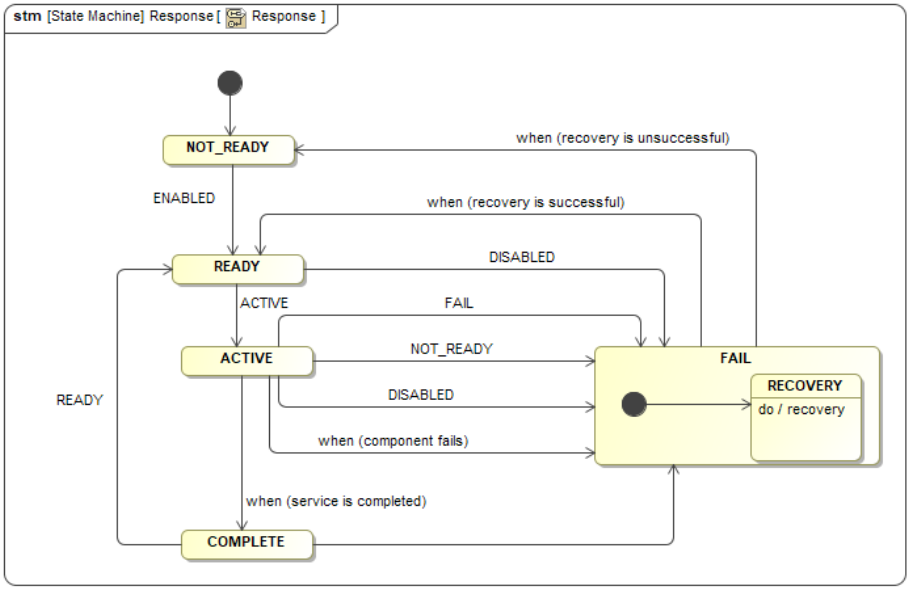

Figure 5: Response State Machine

- `FAIL`

  requester has detected a failure condition.

- `NOT_READY`

  requester is not ready to make a *request*.

- `READY`

  requester is prepared to make a *request*, but no *request* for service is required.

- `RESPONSE`

  operating state of the *response* to a *request* to close a chuck.

  `ResponseStateEnum` Enumeration:

  - `ACTIVE`

    responder has detected and accepted a *request* for a service and is in the process of performing the service, but the service has not yet been completed.

  - `COMPLETE`

    responder has completed the actions required to perform the service.

  - `FAIL`

    responder has detected a failure condition.

  - `NOT_READY`

    responder is not ready to perform a service.

  - READY

    responder is prepared to react to a *request*, but no *request* for service has been detected.

4.3.3 CloseDoor

A subType **MUST** always be specified.

4.3.3.1 Subtypes of CloseDoor

- REQUEST

  operating state of the *request* to close a door.

  The value of CloseDoor **MUST** be one of the RequestStateEnum enumeration.

- RESPONSE

  operating state of the *response* to a *request* to close a door.

  The value of CloseDoor **MUST** be one of the ResponseStateEnum enumeration.

4.3.4 InterfaceState

When the InterfaceState is DISABLED, the state of all data items that are specific for the *interaction model* associated with that Interface **MUST** be set to NOT_READY.

InterfaceStateEnum Enumeration:

- DISABLED

  Interface is currently not operational.

- ENABLED

  Interface is currently operational and performing as expected.

4.3.5 MaterialChange

A `subType` **MUST** always be specified.

4.3.5.1 Subtypes of MaterialChange

- `REQUEST`

  operating state of the *request* to change the type of material or product being loaded or fed to a piece of equipment.

  The value of `MaterialChange` **MUST** be one of the `RequestStateEnum` enumeration.

- `RESPONSE`

  operating state of the *response* to a *request* to change the type of material or product being loaded or fed to a piece of equipment.

  The value of `MaterialChange` **MUST** be one of the `ResponseStateEnum` enumeration.

4.3.6 MaterialFeed

A `subType` **MUST** always be specified.

4.3.6.1 Subtypes of MaterialFeed

- `REQUEST`

  operating state of the *request* to advance material or feed product to a piece of equipment from a continuous or bulk source.

  The value of `MaterialFeed` **MUST** be one of the `RequestStateEnum` enumeration.

- `RESPONSE`

  operating state of the *response* to a *request* to advance material or feed product to a piece of equipment from a continuous or bulk source.

  The value of `MaterialFeed` **MUST** be one of the `ResponseStateEnum` enumeration.

4.3.7 MaterialLoad

A `subType` **MUST** always be specified.

4.3.7.1 Subtypes of MaterialLoad

- `REQUEST`

  operating state of the *request* to load a piece of material or product.

  The value of `MaterialLoad` **MUST** be one of the `RequestStateEnum` enumeration.

- `RESPONSE`

  operating state of the *response* to a *request* to load a piece of material or product.

  The value of `MaterialLoad` **MUST** be one of the `ResponseStateEnum` enumeration.

4.3.8 MaterialRetract

A `subType` **MUST** always be specified.

4.3.8.1 Subtypes of MaterialRetract

- `REQUEST`

  operating state of the *request* to remove or retract material or product.

  The value of `MaterialRetract` **MUST** be one of the `RequestStateEnum` enumeration.

- `RESPONSE`

  operating state of the *response* to a *request* to remove or retract material or product.

  The value of `MaterialRetract` **MUST** be one of the `ResponseStateEnum` enumeration.

4.3.9 MaterialUnload

A `subType` **MUST** always be specified.

4.3.9.1 Subtypes of MaterialUnload

- `REQUEST`

  operating state of the *request* to unload a piece of material or product.

  The value of `MaterialUnload` **MUST** be one of the `RequestStateEnum` enumeration.

- `RESPONSE`

  operating state of the *response* to a *request* to unload a piece of material or product.

  The value of `MaterialUnload` **MUST** be one of the `ResponseStateEnum` enumeration.

4.3.10 OpenChuck

A `subType` **MUST** always be specified.

4.3.10.1 Subtypes of OpenChuck

- `REQUEST`

  operating state of the *request* to open a chuck.

  The value of `OpenChuck` **MUST** be one of the `RequestStateEnum` enumeration.

- `RESPONSE`

  operating state of the *response* to a *request* to open a chuck.

  The value of `OpenChuck` **MUST** be one of the `ResponseStateEnum` enumeration.

4.3.11 OpenDoor

A `subType` **MUST** always be specified.

4.3.11.1 Subtypes of OpenDoor

- `REQUEST`

  operating state of the *request* to open a door.

  The value of `OpenDoor` **MUST** be one of the `RequestStateEnum` enumeration.

- `RESPONSE`

  operating state of the *response* to a *request* to open a door.

  The value of `OpenDoor` **MUST** be one of the `ResponseStateEnum` enumeration.

4.3.12 PartChange

A `subType` **MUST** always be specified.

4.3.12.1 Subtypes of PartChange

- `REQUEST`

  operating state of the *request* to change the part or product associated with a piece of equipment to a different part or product.

  The value of `PartChange` **MUST** be one of the `RequestStateEnum` enumeration.

- `RESPONSE`

  operating state of the *response* to a *request* to change the part or product associated with a piece of equipment to a different part or product.

  The value of `PartChange` **MUST** be one of the `ResponseStateEnum` enumeration.

5 Operation and Error Recovery

The *request and response* state model implemented for *interfaces* may also be represented by a graphical model. The scenario in Figure 6 demonstrates the state transitions that occur during a successful *request* for service and the resulting *response* to fulfill that service *request*.

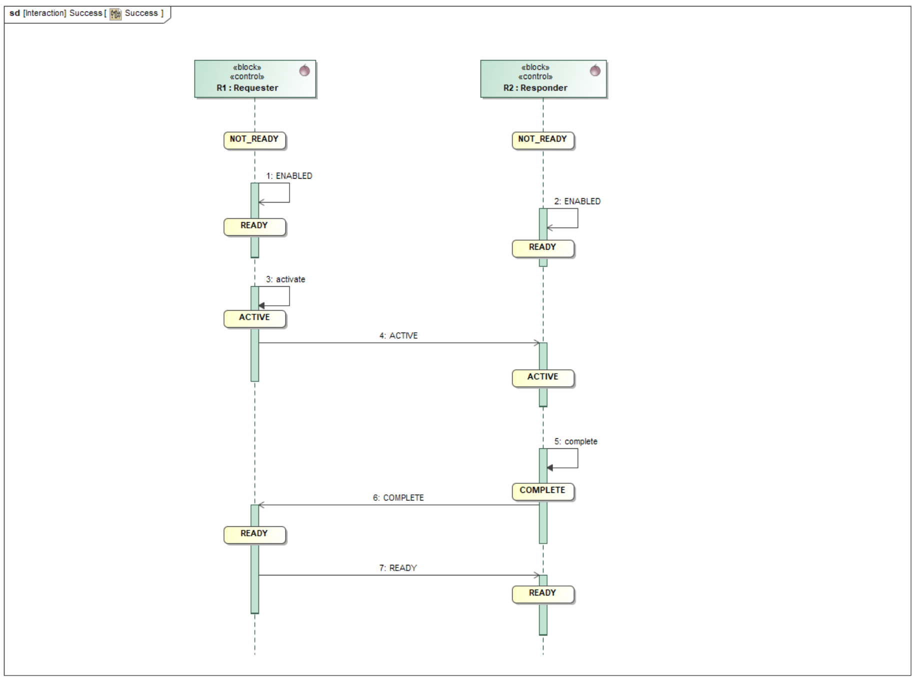

Figure 6: Success Scenario

5.1 Request and Response Failure Handling and Recovery

A significant feature of the *request and response interaction model* is the ability for either piece of equipment to detect a failure associated with either the *request* or *response* actions. When either a failure or unexpected action occurs, the *request* and the *response* portion of the *interaction model* can announce a `FAIL` state upon detecting a problem. The following are graphical models describing multiple scenarios where either the *requester* or *responder* detects and reacts to a failure. In these examples, either the *requester* or *responder* announces the detection of a failure by setting either the *request* or the *response* state to `FAIL`.

Once a failure is detected, the *interaction model* provides information from each piece of equipment as they attempt to recover from a failure, reset all of their functions associated with the *interface* to their original state, and return to normal operation.

The following sections are scenarios that describe how pieces of equipment may react to different types of failures and how they indicate when they are again ready to request a service or respond to a request for service after recovering from those failures:

5.1.1 Responder Fails Immediately

In this scenario, a failure is detected by the *responder* immediately after a *request* for service has been initiated by the *requester*.

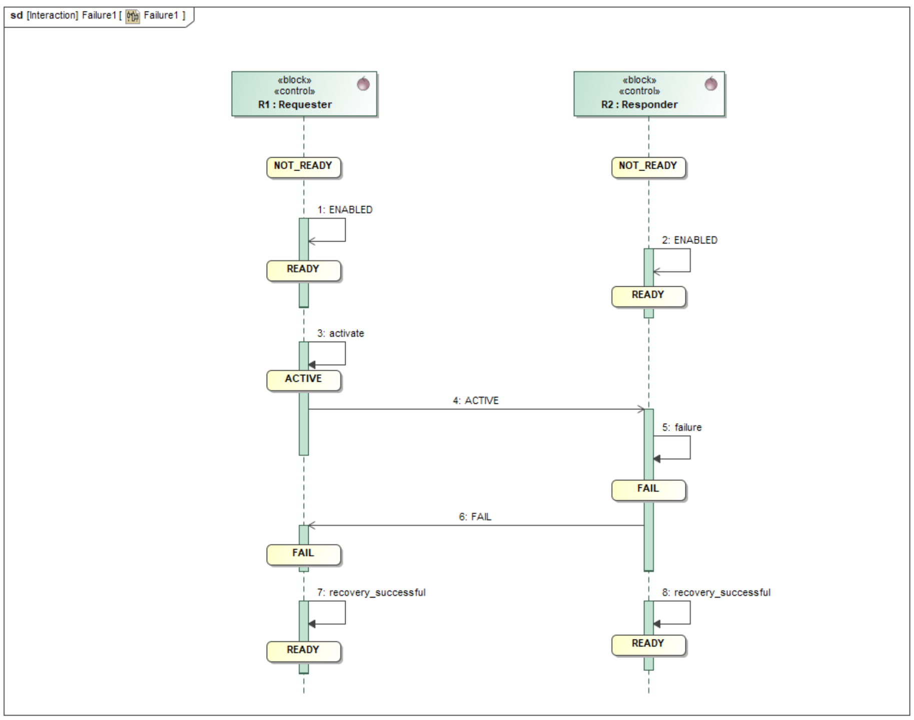

Figure 7: Responder Fails Immediately

In this case, the *request* transitions to `ACTIVE` and the *responder* immediately detects a failure before it can transition the *response* state to `ACTIVE`. When this occurs, the *responder* transitions the *response* state to `FAIL`.

After detecting that the *responder* has transitioned its state to `FAIL`, the *requester* **MUST** change its state to `FAIL`.

The *requester*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *requester* changes its state from `FAIL` to `READY`. If for some reason the *requester* cannot return to a condition where it is again ready to request a service, it transitions its state from `FAIL` to `NOT_READY`.

The *responder*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to perform a service. If the recovery is successful, the *responder* changes its *response* state from `FAIL` to `READY`. If for some reason the *responder* is not again prepared to perform a service, it transitions its state from `FAIL` to `NOT_READY`.

5.1.2 Responder Fails While Providing a Service

This is the most common failure scenario. In this case, the *responder* will begin the actions required to provide a service. During these actions, the *responder* detects a failure and transitions its *response* state to `FAIL`.

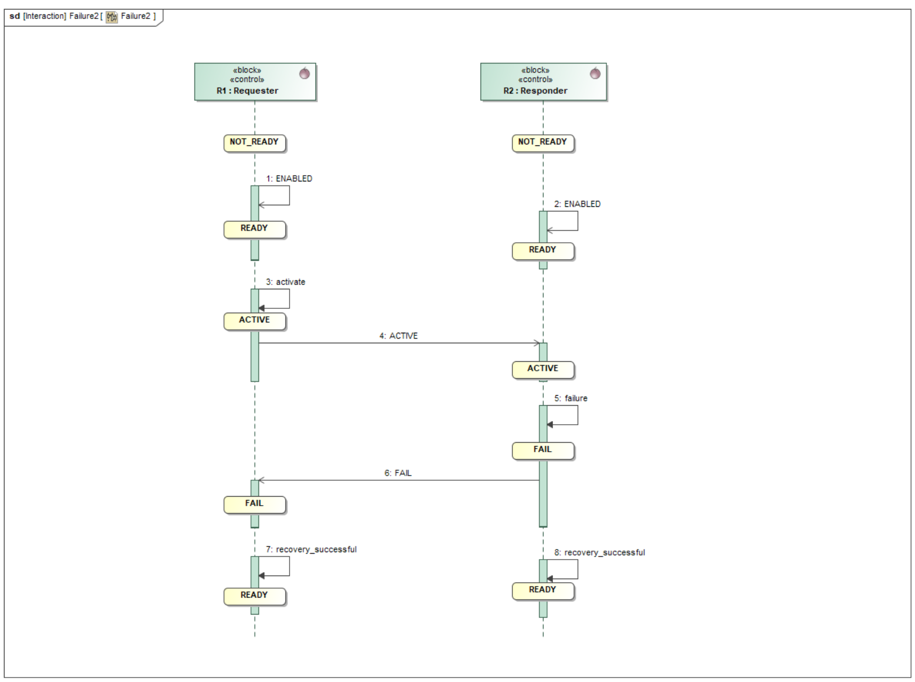

Figure 8: Responder Fails While Providing a Service

When a *requester* detects a failure of a *responder*, it transitions it state from `ACTIVE` to `FAIL`.

The *requester* resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *requester* changes its state from FAIL to READY if the failure has been cleared and it is again prepared to request another service. If for some reason the *requester* cannot return to a condition where it is again ready to request a service, it transitions its state from FAIL to NOT_READY.

The *responder*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to perform a service. If the recovery is successful, the *responder* changes its *response* state from FAIL to READY if it is again prepared to perform a service. If for some reason the *responder* is not again prepared to perform a service, it transitions its state from FAIL to NOT_READY.

5.1.3 Requester Failure During a Service Request

In this scenario, the *responder* will begin the actions required to provide a service. During these actions, the *requester* detects a failure and transitions its *request* state to FAIL.

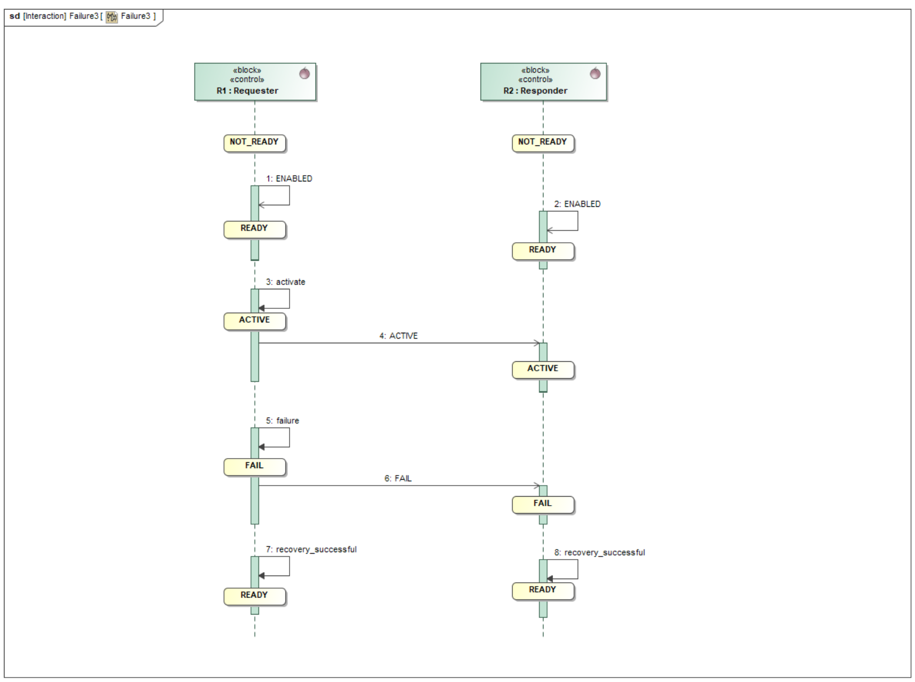

Figure 9: Requester Fails During a Service Request

When the *responder* detects that the *requester* has transitioned its *request* state to FAIL, the *responder* also transitions its *response* state to FAIL.

The *requester*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *requester* changes its state from `FAIL` to `READY`. If for some reason the *requester* cannot return to a condition where it is again ready to request a service, it transitions its state from `FAIL` to `NOT_READY`.

The *responder*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to perform a service. If the recovery is successful, the *responder* changes its *response* state from `FAIL` to `READY`. If for some reason the *responder* is not again prepared to perform a service, it transitions its state from `FAIL` to `NOT_READY`.

5.1.4 Requester Changes to an Unexpected State While Responder is Providing a Service

In some cases, a *requester* may transition to an unexpected state after it has initiated a *request* for service.

As demonstrated in Figure 10, the *requester* has initiated a *request* for service and its *request* state has been changed to `ACTIVE`. The *responder* begins the actions required to provide the service. During these actions, the *requester* transitions its *request* state back to `READY` before the *responder* can complete its actions. This **SHOULD** be regarded as a failure of the *requester*.

In this case, the *responder* reacts to this change of state of the *requester* in the same way as though the *requester* had transitioned its *request* state to `FAIL` (i.e., the same as in Scenario 3 above).

At this point, the *responder* then transitions its *response* state to `FAIL`.

The *responder* resets any partial actions that were initiated and attempts to return to its original condition where it is again ready to perform a service. If the recovery is successful, the *responder* changes its *response* state from `FAIL` to `READY`. If for some reason the *responder* is not again prepared to perform a service, it transitions its state from `FAIL` to `NOT_READY`.

> Note: The same scenario exists if the *requester* transitions its *request* state to `NOT_READY`. However, in this case, the *requester* then transitions its *request* state to `READY` after it resets all of its functions back to a condition where it is again prepared to make a *request* for service.

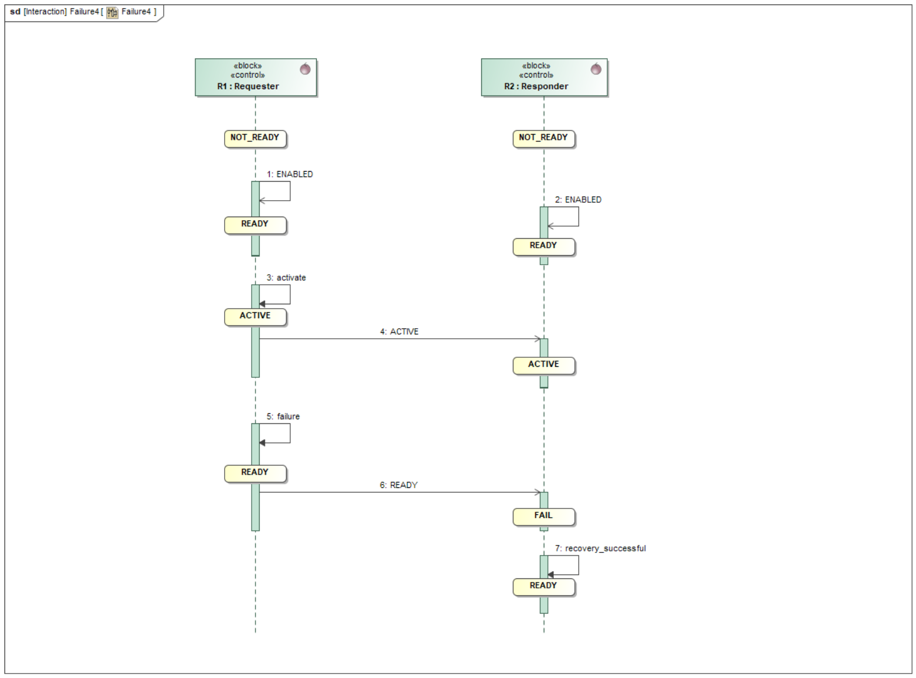

Figure 10: Requester Makes Unexpected State Change

5.1.5 Responder Changes to an Unexpected State While Providing a Service

Similar to Scenario 5, a *responder* may transition to an unexpected state while providing a service.

As demonstrated in Figure 11, the *responder* is performing the actions to provide a service and the *response* state is `ACTIVE`. During these actions, the *responder* transitions its state to `NOT_READY` before completing its actions. This should be regarded as a failure of the *responder*.

Upon detecting an unexpected state change of the *responder*, the *requester* transitions its state to `FAIL`.

The *requester* resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *requester* changes its state from `FAIL` to `READY`. If for some reason the *requester* cannot return to a condition where it is again ready to request a service, it transitions its state from `FAIL` to `NOT_READY`.

Since the *responder* has failed to an invalid state, the condition of the *responder* is un-

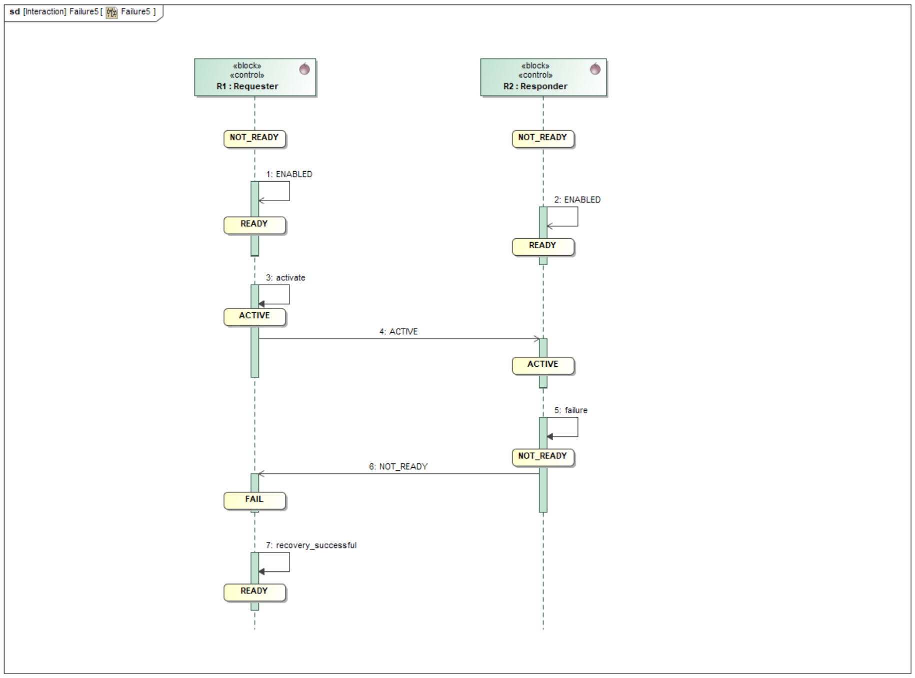

Figure 11: Responder Makes Unexpected State Change

known. Where possible, the *responder* should try to reset to an initial state.

The *responder*, as part of clearing the cause for the change to the unexpected state, should attempt to reset any partial actions that were initiated and then return to a condition where it is again ready to perform a service. If the recovery is successful, the *responder* changes its *response* state from the unexpected state to `READY`. If for some reason the *responder* is not again prepared to perform a service, it maintains its state as `NOT_READY`.

5.1.6 Responder or Requester Become UNAVAILABLE or Experience a Loss of Communication

In this scenario, a failure occurs in the communications connection between the *responder* and *requester*. This failure may result from the `InterfaceState` from either piece of equipment returning a value of `UNAVAILABLE` or one of the pieces of equipment does not provide a heartbeat within the desired amount of time (See *MTConnect Standard Part 1.0 - Fundamentals* for details on heartbeat).

When one of these situations occurs, each piece of equipment assumes that there has been a failure of the other piece of equipment.

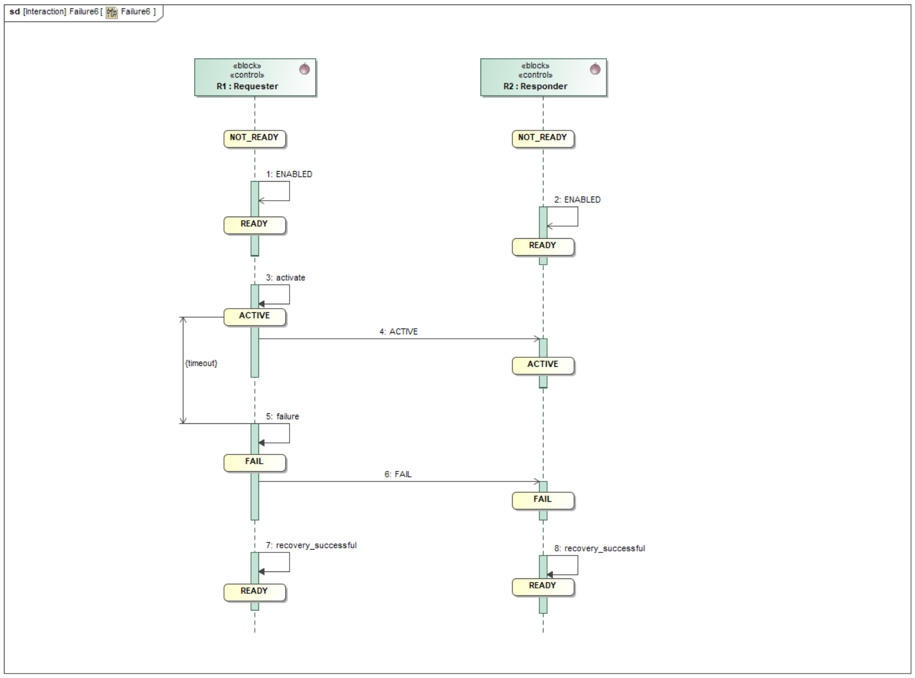

Figure 12: Requester - Responder Communication Failure 1

When normal communications are re-established, neither piece of equipment should assume that the *request and response* state of the other piece of equipment remains valid. Both pieces of equipment should set their state to `FAIL`.

The *requester*, as part of clearing its `FAIL` state, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *requester* changes its state from `FAIL` to `READY`. If for some reason the *requester* cannot return to a condition where it is again ready to request a service, it transitions its state from `FAIL` to `NOT_READY`.

The *responder*, as part of clearing its `FAIL` state, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to perform a service. If the recovery is successful, the *responder* changes its *response* state from `FAIL` to `READY`. If for some reason the *responder* is not again prepared to perform a service, it transitions its state from `FAIL` to `NOT_READY`.

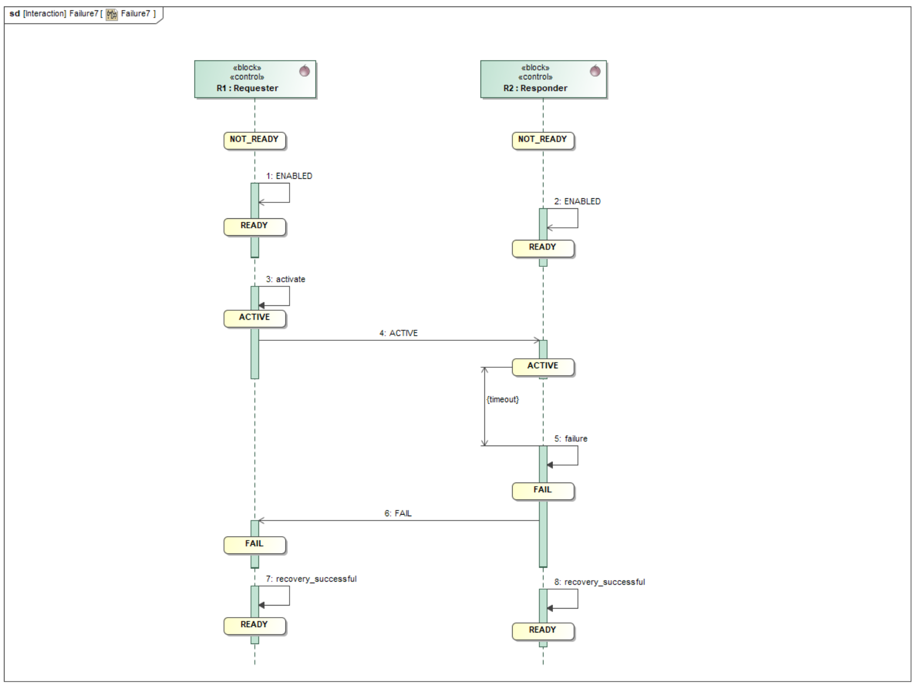

Figure 13: Requester - Responder Communication Failure 2

6 Profile

MTConnect Profile is a *profile* that extends the Systems Modeling Language (SysML) metamodel for the MTConnect domain using additional data types and *stereotypes*.

6.1 DataTypes

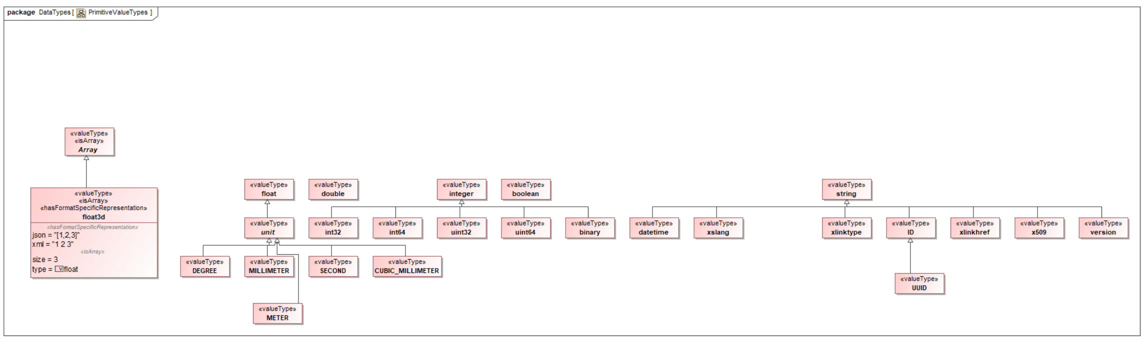

Figure 14: DataTypes

6.1.1 boolean

primitive type.

6.1.2 ID

string that represents an identifier (ID).

6.1.3 string

primitive type.

6.1.4 float

primitive type.

6.1.5 datetime

string that represents timestamp in ISO 8601 format.

6.1.6 integer

primitive type.

6.1.7 xlinktype

string that represents the type of an XLink element. See `https://www.w3.org/TR/xlink11/`.

6.1.8 xslang

string that represents a language tag. See `http://www.ietf.org/rfc/rfc4646.txt`.

6.1.9 SECOND

float that represents time in seconds.

6.1.10 xlinkhref

string that represents the locator attribute of an XLink element. See `https://www.w3.org/TR/xlink11/`.

6.1.11 x509

string that represents an `x509` data block. *Ref ISO/IEC 9594-8:2020*.

6.1.12 int32

32-bit integer.

6.1.13 int64

64-bit integer.

6.1.14 version

series of three numeric values, separated by a decimal point, representing a *major*, *minor*, and *patch* number of the MTConnect Standard.

6.1.15 uint32

32-bit unsigned integer.

6.1.16 uint64

64-bit unsigned integer.

6.1.17 binary

base-2 numeral system or binary numeral system represented by two digits: “0” and “1”.

6.1.18 double

primitive type.

6.1.19 Array

array.

6.1.20 <<hasFormatSpecificRepresentation>>float3d

array of size 3 and datatype float.

6.1.21 UUID

Universally Unique IDentifier. *Ref IETF:RFC-4122*

6.1.22 METER

float that represents measurement in meter.

6.2 Stereotypes

6.2.1 organizer

element that *organizes* other elements of a type.

6.2.2 deprecated

element that has been deprecated.

6.2.3 extensible

enumeration that can be extended.

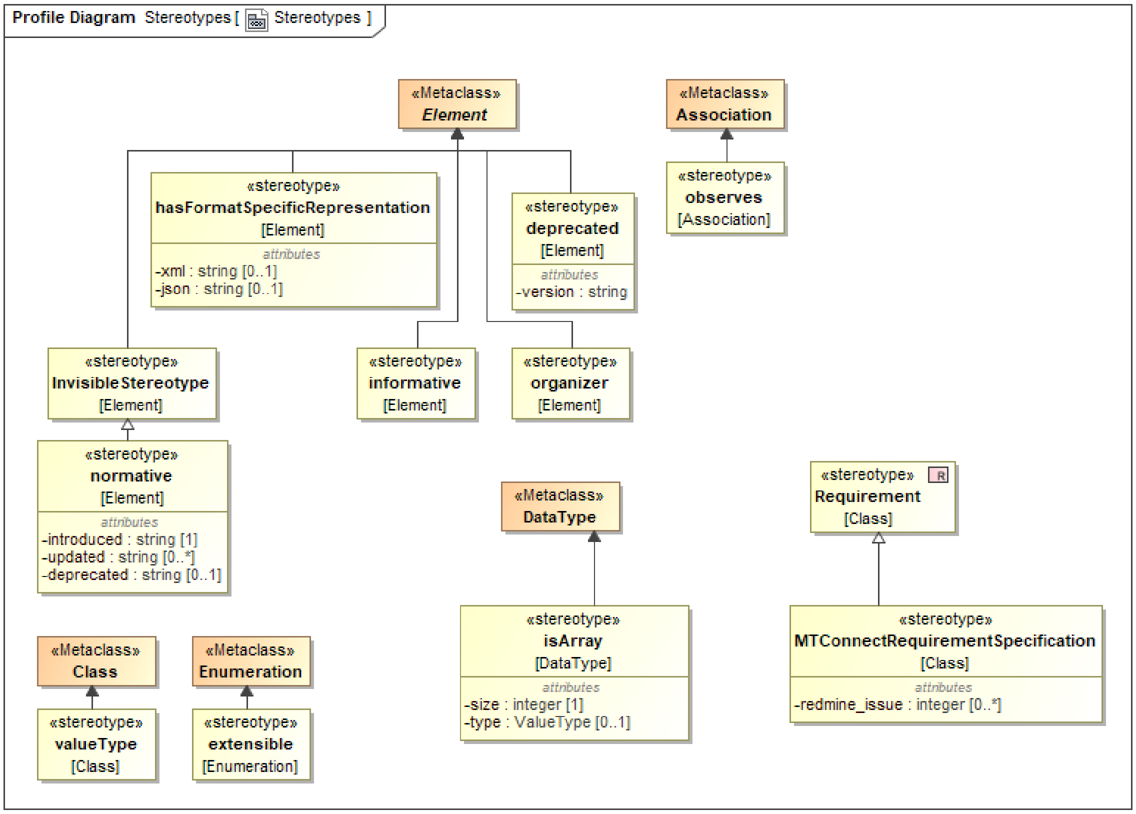

Figure 15: Stereotypes

6.2.4 informative

element that is descriptive and non-normative.

6.2.5 normative

element that has been added to the standard.

6.2.6 observes

association in which a *Component* makes *Observations* about an observable *DataItem*.

6.2.7 satisfiedBy

6.2.8 hasFormatSpecificRepresentation

element that has format specific representation that might be different from the element's SysML representation.

6.2.9 valueType

extends Classto be used as a SysML <<ValueType>>.

6.2.10 isArray

datatype that is an array.

6.2.11 MTConnectRequirementSpecification

MTConnect Requirement.

Appendices

A Bibliography

Engineering Industries Association. EIA Standard - EIA-274-D, Interchangeable Variable, Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines. Washington, D.C. 1979.

ISO TC 184/SC4/WG3 N1089. ISO/DIS 10303-238: Industrial automation systems and integration Product data representation and exchange Part 238: Application Protocols: Application interpreted model for computerized numerical controllers. Geneva, Switzerland, 2004.

International Organization for Standardization. ISO 14649: Industrial automation systems and integration – Physical device control – Data model for computerized numerical controllers – Part 10: General process data. Geneva, Switzerland, 2004.

International Organization for Standardization. ISO 14649: Industrial automation systems and integration – Physical device control – Data model for computerized numerical controllers – Part 11: Process data for milling. Geneva, Switzerland, 2000.

International Organization for Standardization. ISO 6983/1 – Numerical Control of machines – Program format and definition of address words – Part 1: Data format for positioning, line and contouring control systems. Geneva, Switzerland, 1982.

Electronic Industries Association. ANSI/EIA-494-B-1992, 32 Bit Binary CL (BCL) and 7 Bit ASCII CL (ACL) Exchange Input Format for Numerically Controlled Machines. Washington, D.C. 1992.

National Aerospace Standard. Uniform Cutting Tests - NAS Series: Metal Cutting Equipment Specifications. Washington, D.C. 1969.

International Organization for Standardization. ISO 10303-11: 1994, Industrial automation systems and integration Product data representation and exchange Part 11: Description methods: The EXPRESS language reference manual. Geneva, Switzerland, 1994.

International Organization for Standardization. ISO 10303-21: 1996, Industrial automation systems and integration – Product data representation and exchange – Part 21: Implementation methods: Clear text encoding of the exchange structure. Geneva, Switzerland, 1996.

H.L. Horton, F.D. Jones, and E. Oberg. Machinery's Handbook. Industrial Press, Inc.

New York, 1984.

International Organization for Standardization. ISO 841-2001: Industrial automation systems and integration - Numerical control of machines - Coordinate systems and motion nomenclature. Geneva, Switzerland, 2001.

ASME B5.57: Methods for Performance Evaluation of Computer Numerically Controlled Lathes and Turning Centers, 1998.

ASME/ANSI B5.54: Methods for Performance Evaluation of Computer Numerically Controlled Machining Centers. 2005.

OPC Foundation. OPC Unified Architecture Specification, Part 1: Concepts Version 1.00. July 28, 2006.

IEEE STD 1451.0-2007, Standard for a Smart Transducer Interface for Sensors and Actuators – Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats, IEEE Instrumentation and Measurement Society, TC-9, The Institute of Electrical and Electronics Engineers, Inc., New York, N.Y. 10016, SH99684, October 5, 2007.

IEEE STD 1451.4-1994, Standard for a Smart Transducer Interface for Sensors and Actuators – Mixed-Mode Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats, IEEE Instrumentation and Measurement Society, TC-9, The Institute of Electrical and Electronics Engineers, Inc., New York, N.Y. 10016, SH95225, December 15, 2004.