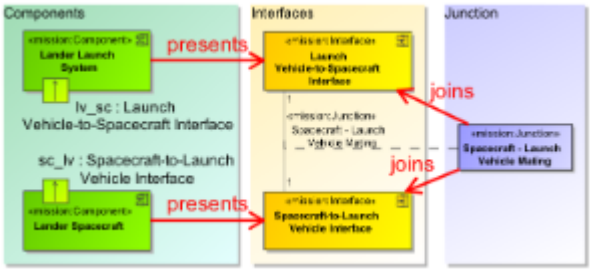
Interface Definition Pattern

Synopsis

This pattern is for enumerating and classifying the allowed or as-designed connections between elements.

Allows users to represent interface types through which energy or material or information flows. Pattern describes what may flow through an interface and allows capture of attributes of those exchanges at a context-agnostic level.



Pattern Overview

Pattern Status	Tool Version
Reviewed	SECAE MagicDraw Packages versions 1702SP3-02 or later

Line Organization Owner	Submitter	Point of Contact
3101 - Engineering Development Office	IMCE Pattern Consolidation Working Group	Dan Dvorak <daniel.l.dvorak@jpl.nasa.gov>

Related Patterns

- Interconnection Pattern: For describing interconnections.
- Structural Context Pattern: For describing the contexts in which interconnection can occur.
- Requirements Definition Pattern: For requirements specification, including specification of interface requirements
- Reconciliation/Abstraction Pattern: For constructing and reasoning across multiple levels of abstraction (logical/physical, conveyance of data across networks, etc.)
- Characterization Pattern: For describing values related to the analysis of an interface. This is useful when the specification is not yet determined and there are trades to be explored.
- Meta-pattern Notes : For constructing and analyzing deployments of your interfaces in specific contexts (scenarios, mission phases, testbeds, etc.)

Note: This pattern is supported by SSCAE MagicDraw Packages versions 1702SP3-02 or later. For more information on the latest SSCAE MagicDraw releases, see the MagicDraw Package page .

Table of Contents

Applicability

To help users assess the applicability of this pattern to their work (i.e., to the problem they want to solve or their area of interest), we describe the way in which this pattern addresses a few kinds of common concerns. In particular, we address:

- Content concerns: the kind of content users can capture in this pattern
- Artifact concerns: the kinds of artifacts (documents and views) that can come from this pattern
- Reasoning concerns: the kind of reasoning (analysis) that this pattern is meant to support
- Assumptions: what we expect to be true about the user's situation that is relevant to whether they can or should use the pattern.

Content Concerns

This pattern provides a mechanism for enumerating and classifying the allowed or as-designed connections between elements. A complete representation of a connection type includes:

- The element types that may interconnect. For semantic clarity, the pattern is restricted to pairwise connections.
- The interface types that may be presented by those element types. Note that an interface is not a physically-realized thing; it is a collection of properties that a pertinent to the connection of one element with another. Note also that an interface is not the connection itself (or the type of the connection).
- The connection types that may join pairs of interfaces.
- The flow and/or item types that may traverse a connection type.
- The quantitative and qualitative values that specify an interface or connection. For example, the specified maximum voltage through a particular power bus, the specified drop rate across an interface, or the specified volume of an interface.

At this generic level, the concern is with specifying the set of connecting elements and available interfaces/junctions. The pattern does not directly provide the means to analyze the consistency of interface definitions across multiple levels of abstraction, but mapping relationships can be layered atop the pattern. The particular use cases for the mapping between layers of abstraction is more easily understood with domain specific examples.

This pattern doesn't assert anything about how to model your specific interface attributes: it addresses only how you might organize your set of interacting elements, how to show their interface presentations, how to connect those interfaces, and how to show what can traverse the junctions. For guidance about capturing properties of the exchanges (data rates, material properties, etc.) you should look to the domain specific examples and/or follow SysML best practices and your best judgement.

It is important to note that the Interface Definition Pattern is concerned with intent at both ends. It talks about components as designed artifacts and interfaces as specified properties of those components. It is applicable when we want to define, or at least account for, the possible exchanges between pairs of interacting components. So it is a natural fit, for example, to the interaction between a spacecraft and a ground system, or between a mission operations system and a ground system. A mission operations system, although it is not a definite physical object, is nonetheless a defined component in the sense that we specify it to perform definite functions and participate in definite interactions. Interface definition is less applicable to the impingement of radiation on a solar panel. One could stretch the pattern to include the Sun as a component, but it would be artificial and unnecessary. A more natural construction would be to consider the solar panel as inhabiting and being affected by a local solar radiation environment, which can be characterized by the relevant fluxes and energies. This construction will be addressed in a pattern to be developed.

Artifact Concerns

This pattern supports the enumeration of content that is present in the following conceptual systems engineering artifacts[note]:

- Functional Block Diagram (FBD)
- Interface Requirements Document (IRD)
- Interface Control Document (ICD)
- Interface specification for mechanical, electrical, and plumbing
- Diagrams showing both sides of each interface (in our terminology, Junction) for a launch vehicle (static and dynamic envelops, spacecraft to launch vehicle, spacecraft to ground support equipment (GSE)).
- (System/subsystem/software) interface lists (SIS)
- System block diagram with interconnections
- DODAF SV-1 Systems Interface Description
- DODAF OV-2 Operational Resource Flow Description
- DODAF SV-2 Systems Resource Flow Description
- DODAF SV-3 Systems-Systems Matrix
- N-squared diagrams

Note: we refer here to the underlying content present in these artifacts, rather than any particular paper examples. This pattern is not "how to

make an IRD;" instead, we assert that this pattern supports the capture of much of the content one would find if one examined many examples of IRDs and retained the core attributes and concerns found therein.

Generic Reasoning Questions

Interface Definition is a set of assertions at the type level about what is allowed. The reasoning one can apply to these assertions is limited; instead these assertions are used primarily to reason about whether an actual set of elements, interfaces, and junctions conforms to what is allowed by Interface Definition.

Examples of Generic Reasoning Questions:

- Are the interface types, presented by each component type, compatible?
  For example, is any usage of the data connection interface type presented by a science instrument appropriate for connecting to the flight computer.
- Are the item and/or flow types, allowed to traverse each junction type, compatible?
  For example, can a certain type of data flow through a particular cable type.

Assumptions

There are currently no assumptions made about the user's situation relevant to this pattern.

Pattern Implementation

Here we describe the elements that make up the pattern, their relationships, responsibilities, and collaborations. The solution does not describe a particular concrete design or implementation. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements solves it. The solution is presented in modeling language independent terms and in its SysML embedding.

Generic/Ontology Implementation

We start with some set of components whose potential to exchange items or information with each other is of interest to us. Exchanges between a pair of candidate components can only happen if both elements possess one side of a compatible pair of interfaces. Interface compatibility is determined in a large part by the ability of both interfaces to convey the kind of item that the systems are intended to exchange. More succinctly, the pair of components present interfaces which may be connected with interface junctions. At this level of abstraction (interface definition), we are saying that these components *may* exercise some connection or exchange something - we say nothing about under what circumstances the connection exists, is available, is exercised, is required, etc. The definition level describes the attributes and characteristics of a theoretical exchange. This allows registry of interfaces to components, exchanged items/messages/flows to interface junctions, connection of interfaces to other interfaces. The interface (or the presentation of the interface by the component) may be specified with requirements - see the [Requirements Pattern] for guidance.

IMCE Concepts

Component

Interface

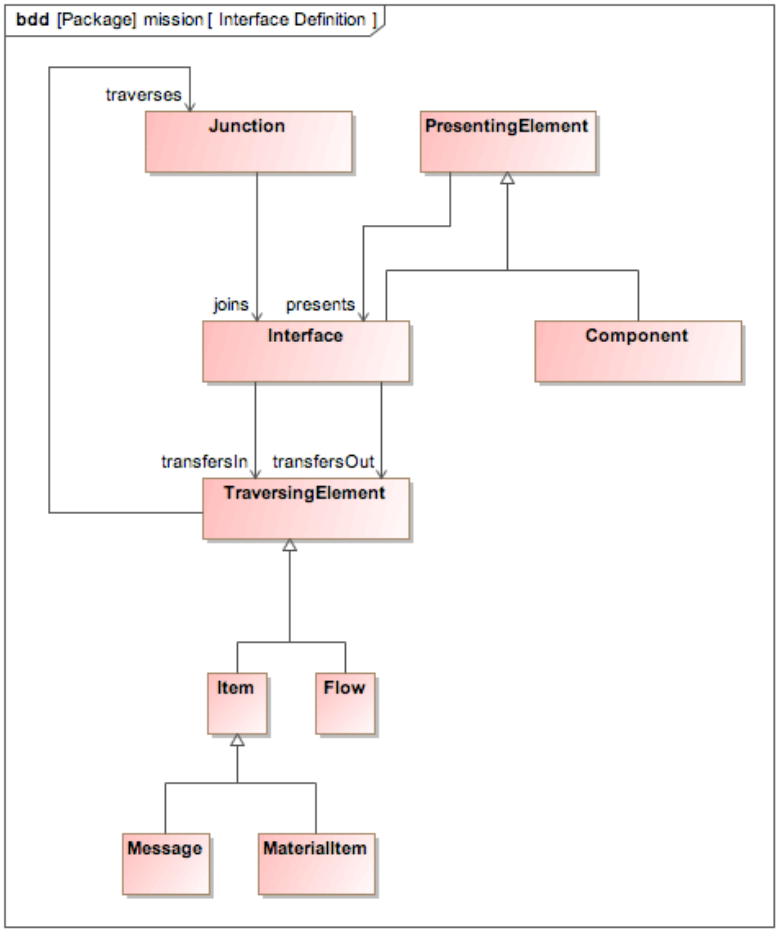
Junction

MaterialItem

Message

Flow

This may be sufficient to describe interfaces for some users. Others, however, may find that they want to talk about how these interfaces are used in scenarios, how they appear across mission phases, how they are exercised in deployment configurations (across many testbed configurations, for example), etc. This is where the Interconnectivity (Interface) Usage Pattern comes in (it will not be described here).



What are these elements? We will describe each of these elements and their intended use in the following paragraphs.

Interface: What kinds of information are captured in an interface?

An **interface** *identifies a set of mechanical, electrical, signal, or other properties that describe some aspect of a component's connection to or interaction with another component.* It is important to distinguish between a component, which is a discrete thing, and an interface, which is a collection of properties. In the formulation phase, for example, we might model the launch vehicle and the spacecraft as components. The properties that describe how the spacecraft is designed to attach to the launch vehicle would be captured in one interface. The properties that describe how the launch vehicle is designed or configured to attach to the spacecraft would be captured in another interface.This method of modeling interfaces allows us to capture the asymmetric aspect of producer/consumer relationships, as well as providing a convenient method for associating the sets of properties and characterize the interconnection of any pair of components so that they can be analyzed for consistency.We define the concept mission:Interface and the relationship mission:Presents to denote the relationship between a component and an interface.

An interface is a container for attributes of the exchange that are necessary in describing how a system connects or exchanges something with other systems. Interfaces describe what a system expects from systems attempting to connect to it; it can also describe what a component "offers" when it connects to other systems. These attributes can be physical, such as mechanical or electrical, or more logical, such as data types, protocols, etc.

Junction: What kind of information do you keep in a junction?

A **junction** *identifies a set of properties that are common to a pair of joined interfaces*. Similarly to an interface, a junction is not a physical entity; but instead a declaration that two interfaces are (under some conditions) joined together and constrained by a set of declared properties.

One important reason for the generic word junction is to avoid premature commitment to any particular implementation approach or technology. In fact, a common pattern is that a junction at one level of abstraction is represented at a lower level of abstraction by an actual component (e.g., a physical cable or harness).

Junctions bind interfaces together to indicate that an exchange between the two components is permitted.

Traversing Elements: What is implied by the assertion that something traverses a junction?

Assertion of interface presentation and assertion of interface joining is not usually sufficient to describe interfaces fully - we need the ability to describe *what* the components are exchanging. Items may be defined and assigned to the Junction to indicate that in addition to other attributes that might be present, interfaces convey and help transport content of interest. Many types of items are available to traverse junctions.

It may be useful at times to distinguish between two subtypes of items: those that constitute matter and those that constitute information. We define mission:MaterialItem as a discrete quantity of matter such as a soil sample and a mission:Message as a discrete unit of information such as a telemetry packet.

Is it an item or a flow?

The key distinction between an item and a flow is primarily the extent of the thing exchanged. An item has a definite extent and is exchanged (at a given level of abstraction) as a discrete, atomic, transaction. A command sequence, for example, is a message (and therefore an item). It has a definite beginning and end, and it makes sense to say it is sent as a single transaction, although at a lower level in the protocol stack it may correspond to a set of transactions. In contrast, flows do not have definite extent. A flow (e.g., electric current) may be switched on or off, but the flow itself has no intrinsic boundaries.

Ultimately, the choice to represent the traversing element as an item or a flow is a modeling choice, where the deciding factors are intent and simplicity. Items are more appropriate when the phenomena of interest involve transaction counts, transaction rates, and temporal precedence. Flows are more appropriate when they underlying phenomena involve continuous-time (e.g., differential) equations. Choose the representation (item or flow) which best facilitates your modeling and analysis.

SysML Implementation

The concepts we described in the last section are mapped to concrete implementation in SysML so that they can be used to actually define interfaces in a model. In this section we describe first the embedding of the ontology into SysML (so that the user can understand how the concepts are made concrete) and then provide examples using a Spacecraft - Launch Vehicle interface to illustrate application of the concepts to a familiar territory.

Embedding

The following table describes how the elements in the ontology appear in SysML. The easiest form of mapping is when one ontological concept is represented by one SysML element: for example, an interface as described in the ontology is mapped to an interface element in SysML with a «mission:Interface» stereotype. However, one-to-one mapping is not always possible, or even the best mapping.

Ontology	Classification	SysML	Stereotype
Interface	Concept	Interface Block	«mission:Interface»
Junction	Concept	Association Block	«mission:Junction»
Component	Concept	UML Component	«mission:Component»
Presents	Relationship	Ownership of a proxy port typed by the interface	n/a
Joins	Relationship	Association Block ends constitute joining	n/a
Traverses	Relationship	Flow properties on Association Block stereotyped by Junction	n/a
Transfers	Relationship	Flow properties on Interface Block stereotyped by Interface	n/a

Concept Mapping: Mapping between Interfaces, Junctions, and Components are straightforward and one-to-one; a user can use existing SysML elements (Blocks, Association Blocks, Components) and apply the correct stereotypes.

Relationship Mapping: As mentioned previously, the mapping between concept and SysML element isn't always one-to-one:

- For **Presents**, there isn't an explicit relationship in SysML between a block and interfaces it presents. To capture the concept of a mission:Component presenting a mission:Interface, we assert that a mission:Component owning a Proxy Port that is typed by a mission:Interface constitutes that mission:Component "presenting" that Interface. In SysML 1.3, there are two types of ports: Proxy Ports and Full Ports. Full Ports represent a component on the boundary of another component. Proxy ports on the other hand represent an interface or connection point. Based on this, a Proxy Port is the appropriate embedding for the Presents relationship.
- For **Joins**, attach an AssociationBlock to exactly two other InterfaceBlocks stereotyped by «mission:Interface», stereotype the AssocationBlock by «mission:Junction». This constitutes the Joins relationship between those elements.
- For **Transfers**, create a FlowProperty owned by a mission:Interface, type this FlowProperty with a mission:TraversingElement. The FlowProperty implies the Transfers relationship. The direction of the FlowProperty must be either In, Out, or InOut. If the direction is In, this asserts that the flow travels into the mission:Component through the mission:Interface.
- For **Traverses,** create a FlowProperty owned by mission:Junction, type this FlowProperty wit ha mission:TraversingElement. The direction of the FlowProperty must be InOut.

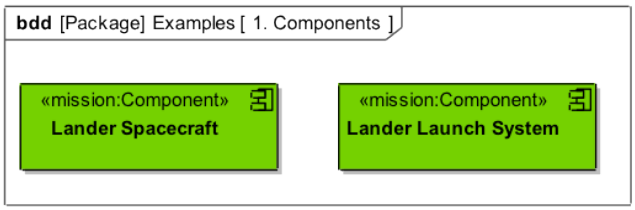
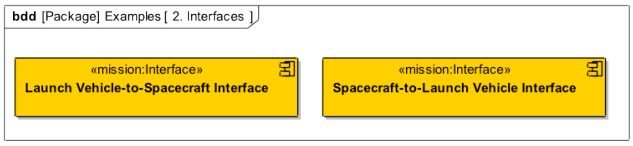
Notes about usage of Traverses vs Transfers:

1. Using a mission:Traverses relationship on a mission:Junction without any mission:Transfers relationships provides a constraints of what can flow through the mission:Junction without capturing the direction or what can flow over the mission:Interface.
2. At runtime, the FlowProperty stereotyped by mission:Traverses provides a reference or probe into what is flowing through the mission:Junction.
3. ValueProperties and ConstraintProperties owned by the Junction allow assertions about the properties of the Junction.
4. Multiple mission:Junctions are allowed between the same mission:Interfaces; these mission:Junctions can have different ValueProperties or ConstraintProperties. A mission:Interface can only have one FlowProperty.

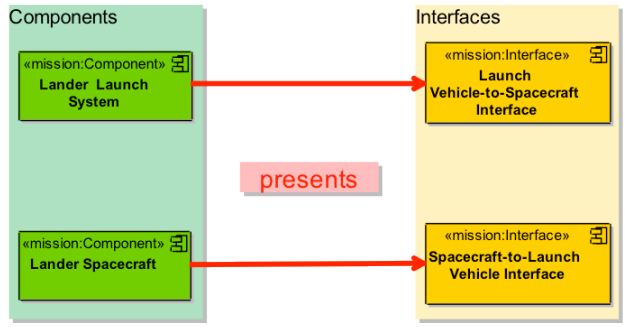
SysML Embedding: In order to make the concepts work correctly, the concepts must be "embedded" into SysML as stereotypes which extend and inherit from UML and SysML metaclasses and existing stereotypes. The SysML Embedding Details section contains diagrams illustrating the "pedigree" of «mission:Component», «Mission:Junction», and «mission:Interface» (and a few other concepts).

Examples - Spacecraft / Launch Vehicle

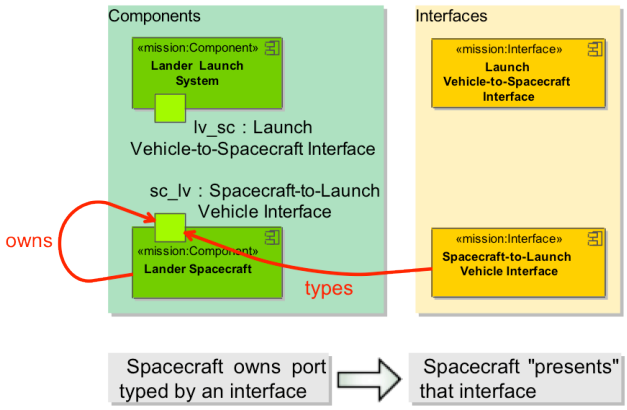
To illustrate the usage of the SysML embedding of the previously described ontology, we will use the pattern to describe an exchange of telemetry on the interface between a Spacecraft and a Launch Vehicle. We want to show the two elements, the interfaces they present, how the interfaces are joined, and how we show telemetry traversing the junction:

Example Image	Description
	We will first show the spacecraft and the Launch Vehicle, both stereotyped with «mission:Component». Note: To create the «mission:Component», it is easiest to create UML Components in the containment tree, stereotype them with «mission:Component», and then drag them onto the BDD. Why? UML Components are not available in the BDD "create-on-diagram" context menu. You may be tempted to create a Block, remove the Block stereotype, and re-stereotype it as a «mission:Component», but that will not be correct because the metaclass will be Class instead of Component.
	We next define two interfaces: the LV-to-SC interface and the SC-to-LV interface. Note: LV-to-SC doesn't mean that LV is providing anything to the SC; rather, it is a naming convention that indicates that this "half" of the interface is owned by the LV. Currently, «mission:Interface» is based on Component. We assert here that a «mission:Interface» specializes «InterfaceBlock», which is not possible if it's based on Component.

"Metaphorical" Diagram: This is what we want to show! How do we do it?

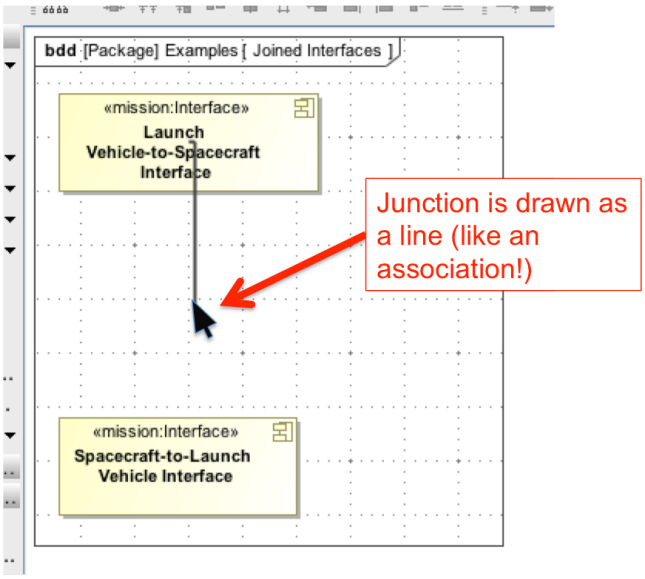


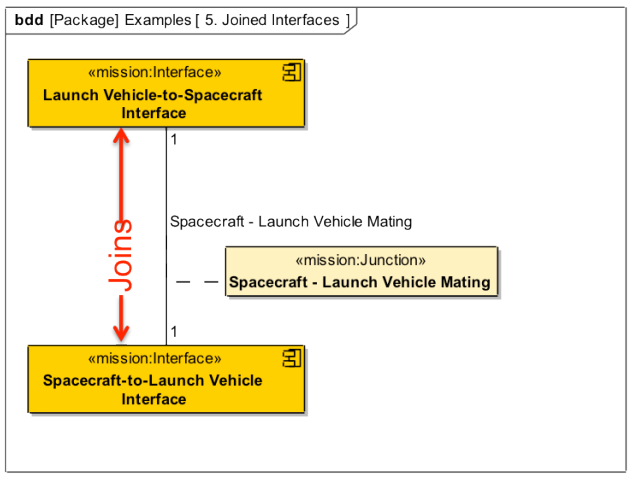
Now we want to say that our components present these interfaces. How do we do it? At left you see a logical presents relationship - this is the content we want to express. How do we do it?
As we described previously, "presents" is asserted by our component owning a port which is typed by the interface we want to present.
So, we'll have to make a port, and type it by the interface.



The ***presents*** relationship is represented by a **Component** *owning* a **UML::port** *typed* by an **Interface**.

We've created a port on each component, named lv_sc and sc_lv respectively. The red arrows show (metaphorically) the ownership and typing that together assert the presentation of the interface.





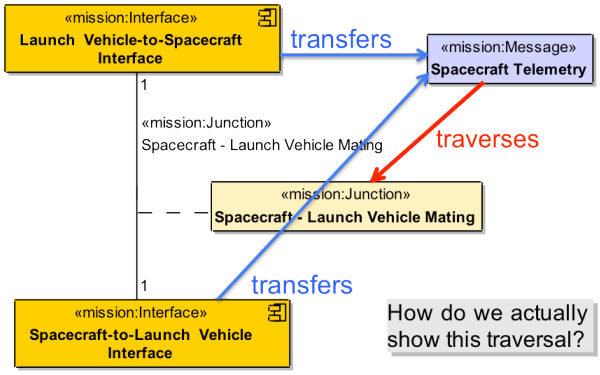
Now that we've defined our interfaces and components, how do we talk about the exchange of information between the SC and LV?

To do that, we join the interfaces with an interface junction (at left you see a screenshot where a modeler is part way through drawing the Junction between the two interfaces). The interface junction is drawn like an association, but when it is created, looks like the second picture on the left.

Now we have asserted that the SC and LV may communicate somehow (we haven't defined what or how they do that) through the junction. This also asserts the "joins" relationship between the junction and each interface.

Note: currently, binary interfaces (i.e., pairs of interfaces) is the only interface configuration supported by this pattern. A junction may join exactly two interfaces

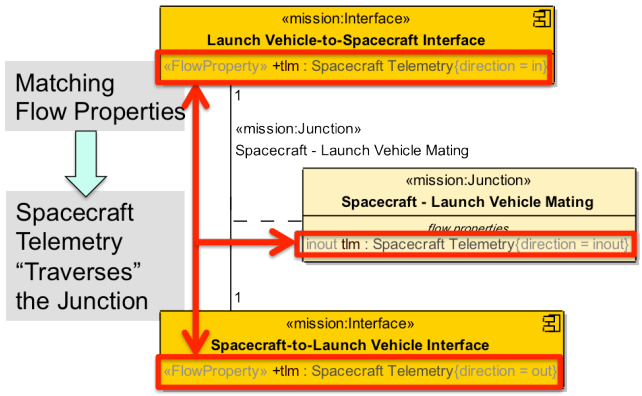
"Metaphorical" Diagram: This is what we want to show! How do we do it?



Now we want to talk about what traverses the junction. For our example, we want to look at Spacecraft Telemetry that is given to the LV by the SC (during ascent, for example, SC telemetry is interleaved with LV telemetry since the SC is not available for direct communication while inside the LV, as direct communication through the umbilical is broken as the umbilical detaches on liftoff).

As discussed previously, there is an ontological relationship between some kind of TraversingElement and the junction ("traverses"). There is also a relationship between the interfaces and the TraversingElement ("transfers").

Rather than make three cumbersome relationships between TraversingElements and the pieces of our interface definition, we can create matching flow properties in each element ("matching" means that they have the same type, and compatible direction).



To assert the traversal of Spacecraft Telemetry across the Junction, we create Flow Properties in each interface and the Junction, and type the properties with Spacecraft Telemetry.

We want to show that the telemetry leaves the SC and goes to the LV; we do this by setting the direction attribute on the property to "out" on the SC's interface (SC-to-LV) and "in" on the LV's interface. The Junction's property direction is "inout" (since it wouldn't be in or out, rather both, as it traverses the junction).

It might appear that the assignment of flow properties to interfaces and junctions is redundant. For example, the fact that these interfaces exchange Spacecraft Telemetry would appear to imply that the flow property of the junction must also be Spacecraft Telemetry, but this is not strictly the case. The actual requirement is that the junction flow property must be the same type as those of the joined interfaces *or some supertype of it*. As libraries of reusable model elements continue to grow, it will be common to reuse definitions of interface pairs and junctions as a set with specialization of the interface types. For example, the type of the interface flow property could be Cubesat Telemetry (a specialization of Spacecraft Telemetry) in a specific application without any requirement to similarly specialize the junction.

Rules/Axioms/Invariants

ID	Restriction Rule
JPL-SysML-04	All SysML Ports must have isService=true
Onto-Block-01	All SysML Blocks must be either a kind of Mission:Component or Mission:MaterialItem, both of which must have homogeneous structure in terms of Mission:Component and Mission:MaterialItem respectively
Onto-Block-03	All Mission:Junctions (i.e., SysML AssociationBlocks) must be directed in the sense that the concrete extent of the joined Mission:Interface (i.e., SysML InterfaceBlock) must be disjoint.
Onto-Port-01	ProxyPorts cannot be conjugated
Onto-Property-01	Any SysML flow property typed by a Mission:Item or Mission:Flow must have composite aggregation
Onto-Property-02	FlowProperties can only be typed by a Mission:Item or Mission:Flow
Onto-Property-03	Use a value property typed by a SysML value type owned by an Interface or Junction as a characterization of the interaction involved in communication through the Interface or Junction. Such a characteristic value property of an Interface or Junction can be accessed inside or outside the scope of the part owning the proxy port using binding connectors.
Owner-01	AssociationBlocks can only have connector, constraint, flow, part, participant, reference and value properties
Owner-02	Class Blocks can only have connector, constraint, part or reference properties and full and proxy ports
Owner-04	InterfaceBlocks can only have connector, constraint, flow, reference and value properties and proxy ports
Owner-05	Signals can only own value properties
Owner-06	ValueTypes can only own value properties
Owner-07	Only SysML StructuredBlocks, i.e., Class or Interface Blocks, can have SysML Ports (Full or Proxy)

Owner-08	An interface block can only own a proxy port OR some combination of Value, Flow, and Reference properties
Typing-02	A ConstraintParameter can only be typed by a Signal or ValueType
Typing-03	A ConstraintProperty must be typed by a ConstraintBlock
Typing-05	A ParticipantProperty can only be typed by a ClassBlock
Typing-06	A PartProperty can only be typed by a ClassBlock
Typing-07	Proxy ports must be typed by interface blocks
Typing-08	A ReferenceProperty can only be typed by a ClassBlock
Typing-09	A ValueProperty can only be typed by a ValueType
Workaround-01	InterfaceBlock can have only 1 Flow or Reference property

Model Implementation Concerns

Nested Ports:

- When nesting ports you should be clear whether you are doing it for visual/layout reasons or whether you are trying to express some semantics. If you choose nesting, you should be able to articulate the semantic you are trying to express. Key issue is that when you nest connections/interfaces, reuse is limited to the context in which you defined it: for example, if you transmit 25 kinds of data, you can only reuse lower level connections for those 25 kinds of data, not for participating in exchanges between other systems (router to router, e.g.).

Decomposition of connectors:

- For example, nested port can imply a decomposition of interfaces, like an umbilical containing many cables (all transporting different things). What does decomposing a connector mean? Aggregation of other connectors, but generally does not imply something like a protocol stack.

Validation/Well-Formedness Reasoning

Assertions to be validated:

- The type of the item flow must be the same or a specialization of the type of the flow properties.

Supporting Scripts/Tooling

This pattern is supported by SSCAE MagicDraw Packages versions 1702SP3-02 or later. For more information on the latest SSCAE MagicDraw releases, see the MagicDraw Package page.

Name	Description	Status	Author/Provider	Location
Pattern Validator	MagicDraw plugin that checks pattern construction in projects	Complete for Interface Definition Pattern Validation	mjackson	JPatterns Project in Stash
	Creates Junction/Interface/ Properties sets based on item flows	In Work	mjackson	TBD
	Creates Junction/Interface/ Properties based on usage pattern	Complete, possibly out of date	mjackson	TBD

Tooling Tricks

Trick	How to do it	Why?
Junctions should own the ends of the association	Open the element specification for the association class («Junction»). Scroll down to the property sets for the association ends ("Role of ...") and locate the "Owned By" property. This property is editable. Select "Block (<block name>)."	This allows reuse of interfaces across modules. By default when an association is created between blocks, the properties are owned by the blocks. This is not allowed if one block is in a read-only module, since the property cannot be created.

Open Questions

- Nested Ports? (Can you use them? Are they useful? What if you already have them?)
- UML ball-and-socket ports? (Can you represent one-to-many (multicast, GPS, services, etc.) this way? What if you already have?)
- Are there "structural presentation relationships" that users have to find and stereotype? Might be a lot of overhead?
- Package Structure - where should interfaces reside in the model?
- What can be said in the definition of interfaces to describe things like "separation"?
- How can item/information flows be utilized / related to this pattern? Can they be used "before" interface definition? How to make it "legal": want to show at BDD level the item exchanges without defining interfaces yet - how to do this and make it valid? Acceptable amount of clicks and extra properties?
- This pattern does not cover interactions between function. How to distinguish between "functional" or "logical" elements and actual functions?

Pattern Resources

References

Currently no Working Group approved references. See the Community Page: Interface Definition Pattern references area for unofficial references.

Modeling Guide (somewhat outdated): SysML Modeling Guide

Further Examples

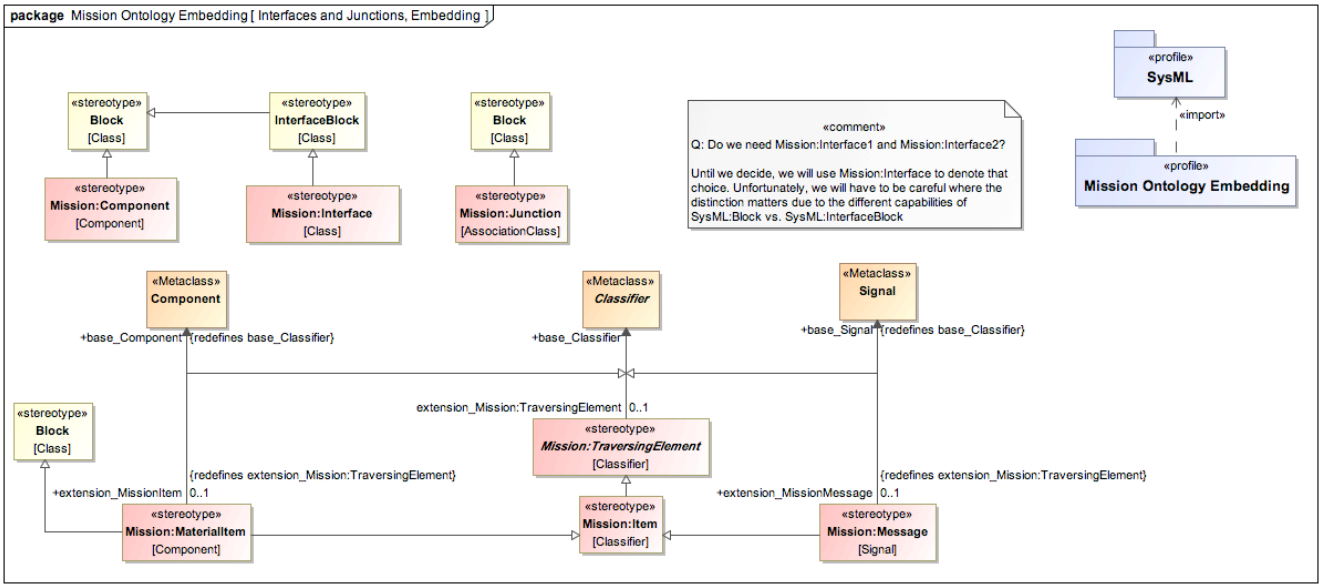
Currently no Working Group approved examples. See the Community Page: Interface Definition Pattern examples area for further examples.

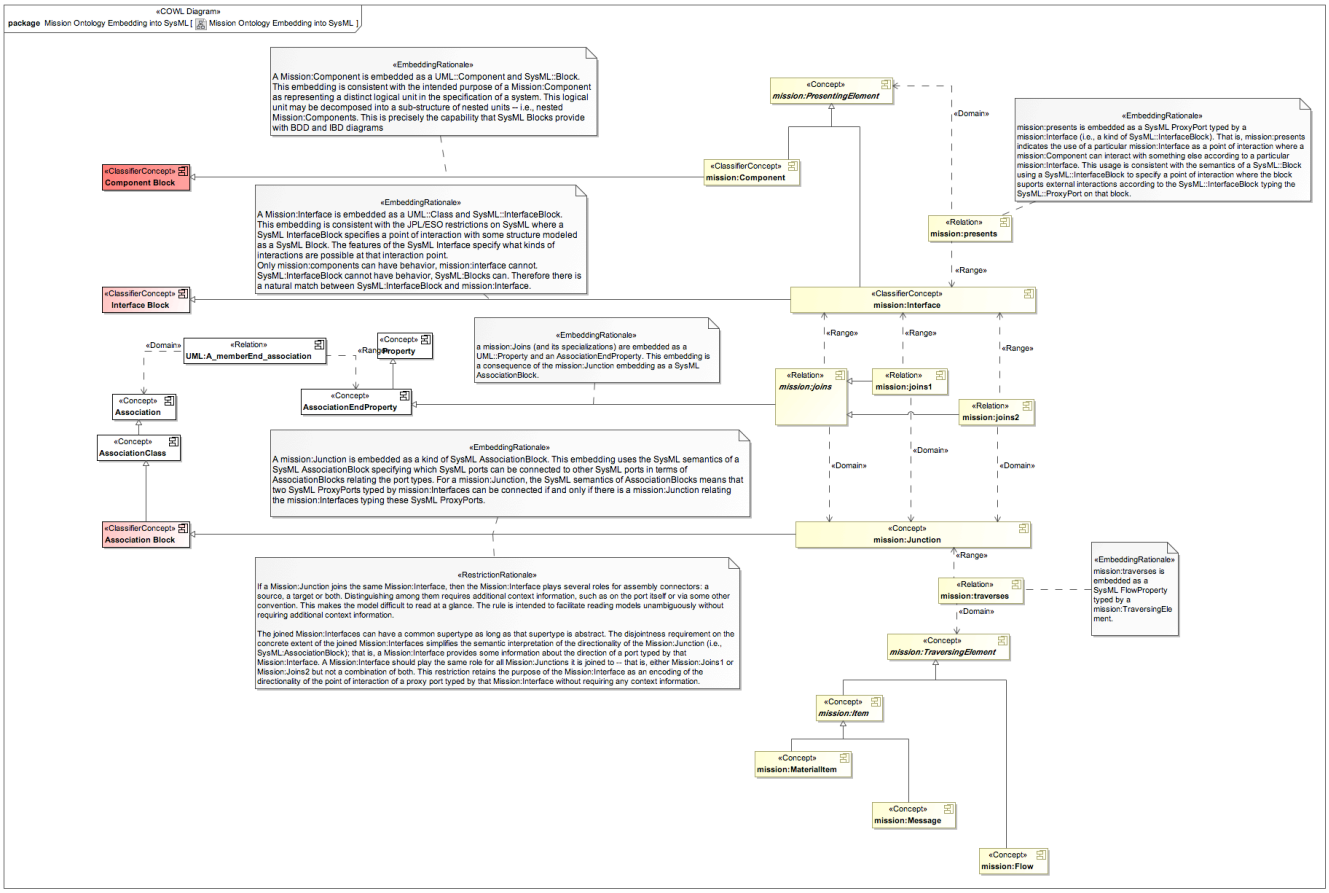
Community Page

The Community Page: Interface Definition Pattern has been set up to collect Frequently Asked Questions, Discipline Specific (and extended) examples and reasoning, and References. Everyone should have write access and are free to discuss and contribute.

Appendix

SysML Embedding Details





We can see from this image that a «mission:Component» is a specialization of a SysML «Block» and is also based on a UML Component (note the word Component in brackets). The "[Component]" seen below the stereotype name is extension. The line ending in a black-filled arrowhead is also extension; the presence of the relationship on the diagram is optional (note that the extended element in brackets is present regardless of the presence of the arrow, and matches the arrow if present).

We see also that «mission:Interface» is a type of SysML «InterfaceBlock», which itself is a specialization of SysML «Block». So «mission:Interface» inherits all of the attributes of a Block and an InterfaceBlock from SysML, which a «mission:Component» inherits the attributes of Block, but will have the appearance and behavior also of a UML Component.

«mission:Junction» is also a kind of block, but extends Association Class. (An Association Class is the UML version of Association Block, which is both a Block and an Association.)

We also see «mission:TraversingElement» described here. The most general type of TraversingElement is an Item, with specializations for Messages (which extend Signal) and MaterialItems (which extend UML Components and are also Blocks).

Copyright