Introduction

This document specifies the metamodel for IFML Diagram Interchange (IFML DI). The IFML DI is meant to facilitate interchange of IFML diagrams between tools rather than being used for internal diagram representation by the tools.

The IFML DI metamodel is defined as a MOF-based metamodel. The IFML DI classes only define the visual properties used for depiction. All other properties that are required for the unambiguous depiction of IFML diagram elements are derived from the referenced IFML model elements.

Multiple depictions of a specific IFML Element in a single diagram are not allowed.

Architecture

The IFML language specification provides three normative artifacts at M2 (shown with shaded boxes in Figure 1): the abstract syntax model (IFML), the IFML diagram interchange model (IFML DI), and the mapping specification between the IFML DI and the graphics model (IFML Mapping Specification).

At M1 (left), Figure 1 shows an instance of IFML::Core::ViewContainer as a model element. Next to it, on the right, the figure shows an instance of IFMLDI::IFMLNode referencing the ViewContainer element, indicating that the ViewContainer is depicted as a node on the diagram.

Figure 1: Diagram Definition Architecture for IFML
The node also contains an instance of IFMLDI::IFMLLabel representing the textual label of the ViewContainer on the diagram. On the right of M1, the figure shows an instance of DG::Group containing instances of DG::Rectangle and DG::Text.

IFML DI specializes DD:DI, which specifies the graphics the user has control over, such as the position of nodes and line routing points. This information is what is captured for interchange between tools.

DD:DG represents the graphics that the user has no control over, such as shape and line styles, because they are the same in all languages conforming to the DD specification. DD:DG is derived by executing the mapping specification, in the middle, between IFML DI and DG.

### IFML Diagram Interchange (DI) Meta-model

The IFML DI metamodel extends the DI metamodel, where appropriate. The class IFMLDiagram represents the diagram, which composes IFMLDiagramElements. An IFMLDiagram is an IFMLNode because it may be rendered as a figure and be connected to other figures.

IFMLDiagramElements optionally reference elements of an IFML model, the latter denoted by the IFML::Core::Element class. IFMLDiagramElements that do not reference elements of an IFML model are purely notational diagram elements such as notes and the link that connects the note with the model element. IFMLDiagramElements may also be styled with instances of class IFMLStyle (e.g., font type and size).

**Figure 2: IFML Diagram Interchange (DI) Meta-model**

Classes are defined for interchanging shapes and edges of the interaction flow diagram and the content diagram, based on the following notational patterns (see Figure 3):

- Pattern (a): A shape that has a label and an optional list of compartments, each of which having an optional list of labels or other shapes (e.g., the ViewContainer box, ViewComponentPart box, Form ViewComponent rounded box or the classes of the
ContentModel).

- Pattern (b): A shape that has a label only (e.g., the Event ball or Action hexagon notation)
- Pattern (c): An edge that may be dashed or solid (e.g., NavigationFlows and DataFlows)

![Diagram with patterns](image)

**Figure 3: Notational patterns**

Based on these patterns, three shape classes (IFMLNode, IFMLLabel and IFMLCompartment) and one edge class (IFMLConnection) are defined and related to realize the patterns. These classes (except IFMLCompartment) are subclasses of IFMLDiagramElement to allow them to be styled independently and to reference their own IFML Element.

Some classes have properties to disambiguate the notation and a corresponding enumeration. For instance labels may be of different kinds such as Parameter, ViewContainer, etc., which will determine how the text decoration will be rendered.

**IFML DI to DG Mapping Specification**

The DD architecture expects language specifications to define mappings between interchanged and non-interchanged graphical information, but does not restrict how it is done. The IFML DI to DG mapping is accomplished in this specification by means of the following QVT mapping.

```plaintext
transformation IFMLDItoDG(in ifmldi: IFMLDI, in ifml: IFML, out DG)
main()
{
  ifmldi.objectsOfType(IFMLDiagram)->map toGraphics();
}
```
mapping IFMLDiagram::toGraphics(): Canvas {
    member += self.diagramElements->map toGraphics();
}

mapping IFMLDiagramElement::toGraphics(): Group {
    localStyle := copyStyle(self.localStyle);
    sharedStyle := copyStyle(self.sharedStyle);
}

mapping IFMLNode::toGraphics(): Group inherits IFMLDiagramElement::toGraphics() {
    member += self.modelElement.map toGraphics();
    member += self.ownedCompartments->map toGraphics();
    member += self.ownedLabel.map toGraphics();
}

mapping IFMLLabel::toGraphics(): Text inherits IFMLDiagramElement::toGraphics() {
    var e := self.modelElement;
    bounds := self.bounds;
    data := switch {
        case (self.kind = LabelKind::NAMED_ELEMENT) e.name;
        case (self.kind = LabelKind::VIEW_CONTAINER)
            e.oclAsType(ViewContainer).getLabelText();
        case (self.kind = LabelKind::ACTION)
            e.oclAsType(Action).getLabelText();
        case (self.kind = LabelKind::PARAMETER)
            "Parameter " + e.name + ": " + e.type.name;
        case (self.kind = LabelKind::ENTRY)
            "Entry " + e.name;
        case (self.kind = LabelKind::LIST)
            "List " + e.name;
        case (self.kind = LabelKind::SIMPLE_FIELD)
            "SimpleField " + e.name;
        case (self.kind = LabelKind::SELECTION_FIELD)
            "SelectionField " + e.name;
        case (self.kind = LabelKind::PARAMETER_BINDING_GROUP)
            "ParameterBindingGroup";
        case (self.kind = LabelKind::PARAMETER_BINDING_GROUP)
            "ParameterBindingGroup";
        case (self.kind = LabelKind::PARAMETER_BINDING_GROUP)
            "ParameterBindingGroup";
        case (self.kind = LabelKind::PARAMETER_BINDING_GROUP)
            "ParameterBindingGroup";
        case (self.kind = LabelKind::ACTIVATION_EXPRESSION)
            "ActivationExpression";
        case (self.kind = LabelKind::INTERACTION_FLOW_EXPRESSION)
            "InteractionFlowExpression";
        case (self.kind = LabelKind::VALIDATION_RULE)
            "ValidationRule";
        default "";
    }
};

query ViewContainer::getLabelText(): String {
    var text := if self.isXOR then "[XOR] " endif;
    text += if self.isLandmark then "[L] " endif;
    text += if self.isDefault then "[D] " endif;
    return text + self.name;
}

query Window::getLabelText(): String {
    var text := if self.isNewWindow and self.isModal then "[Modal] " endif;
    text += if self.isNewWindow and not self.isModal then "[Modeless] " endif;
    text += if self.isLandmark then "[L] " endif;
    text += if self.isDefault then "[D] " endif;
    return text + self.name;
}

query Action::getLabelText(): String {
    var text := if self.isClientSide then "[ClientSide]\n" endif;
    return text + self.name;
}

mapping Element::toGraphics(n: IFMLNode): GraphicalElement
    disjuncts ViewContainer::toRectangle, ViewComponent::toRectangle,
    Module::toRectangle, ViewComponentPart::toRectangle, Event::toCircle,
    Action::toPolygon, ViewPoint::toPolygon { n.bounds := n.bounds;
}

mapping ViewComponent::toRectangle(n: IFMLNode): Rectangle {
bounds := n.bounds;
cornerRadius := 15;
}

mapping Module::toRectangle(n: IFMLNode): Rectangle {
bounds := n.bounds;
}

mapping ViewComponentPart::toRectangle(n: IFMLNode): Rectangle {
bounds := n.bounds;
}

mapping Event::toCircle(n: IFMLNode): Circle {
  var b := n.bounds;
  center := object Point{b.x + b.width / 2; b.y + b.height / 2};
  radius := if b.width < b.height then
    b.width / 2
  else
    b.height / 2
  endif;
}

mapping Action::toPolygon(n: IFMLNode): Polygon {
  var b := n.bounds;
  point += object Point {b.width * (1/4); y := 0};
  point += object Point {b.width * (3/4); y := 0};
  point += object Point {b.width; b.height * (1/4)};
  point += object Point {b.width; b.height * (3/4)};
  point += object Point {b.width * (3/4); b.height};
  point += object Point {0; b.height * (1/4)};
  point += object Point {0; b.height * (3/4)};
}

mapping ViewPoint::toPolygon(n: IFMLNode): Polygon {
  var b := n.bounds;
  point += object Point {b.width * (1/2); y := 0};
  point += object Point {b.width; b.height};
  point += object Point {0; b.height};
}

mapping ParameterBindingGroup::toPolygon(n: IFMLNode): Polygon {
  var b := n.bounds;
  point += object Point {x:=0,y:=0};
  point += object Point {b.width*3/4,y:=0};
  point += object Point {b.width,b.height};
  point += object Point {b.width*1/4,b.height};
}

mapping IFMLCompartment::toGraphics(): Group {
  member += object Rectangle {bounds:= self.bounds};
  member += self.ownedNodes.map toGraphics();
  member += self.ownedLabels.map toGraphics();
}

mapping IFMLConnection::toGraphics(): Group inherits IFMLDiagramElement::toGraphics()
{
  member += self.modelElement.map toGraphics(self);
}

mapping Element::toGraphics(c: IFMLConnection): GraphicalElement disjuncts NavigationFlow::toPolyline, DataFlow::toPolyline {
  point := c.waypoint;
  sharedStyle := solidStyleProp;
  endMarker := arrowMarkerProp;
}

mapping NavigationFlow::toPolyline(c: IFMLConnection): Polyline {
  point := c.waypoint;
  sharedStyle := solidStyleProp;
  endMarker := arrowMarkerProp;
}

property solidStyleProp = object DG::Style {
  strokeDashLength := Sequence();
}

property arrowMarkerProp = object Marker {
  size := object Dimension {width := 2; height := 2};
  reference := object Point {x := 2; y := 1};
  member += object Polyline {
    point += object Point {x := 0; y := 0};
    point += object Point {x := 2; y := 1};
    point += object Point {x := 0; y := 2};
  }
}

mapping DataFlow::toPolyline(c: IFMLConnection): Polyline {
  point := c.waypoint;
  sharedStyle := dashedStyleProp;
  endMarker := arrowMarkerProp;
}
property dashedStyleProp = object DG::Style {
    strokeDashLength := Sequence(2, 2);
}

helper copyStyle(s: IFMLStyle): DG::Style {
    fontName := s.fontName;
    fontSize := s.fontSize;
    fillColor := s.fillColor;
}