**The Industrial Kitting Workstation Ontology Version 0.6**

**Craig Schlenoff, Thomas Kramer, Zeid Kootbally, Anthony Pietromartire, and Stephen Balakirsky**

**December 8, 2013**

# INTRODUCTION

The purpose of this paper is to document the contents of the industrial kitting workstation ontology being developed at the National Institute of Standards and Technology (NIST) as part of the Agility Performance of Robotic Systems (APRS) project in the Robotic Systems for Smart Manufacturing program. This ontology will also serve as the basis for the Industrial Robotics Ontology as part of the IEEE Robotics and Automation Society’s (RAS) Ontologies for Robotics and Automation (ORA) Standard Working Group[[1]](#footnote-1).

The goal of the APRS project is to develop the measurement science and standards to assess the agility of robots with respect to changeover times for new tasks and new products. The key idea for this project is to develop the measurement science, in the form of an integrated agility framework, which will enable manufacturers to assess and assure the agility performance of their robot systems. This framework includes robot agility performance metrics, information models, test methods, and protocols – all of which will be validated using a combined virtual and real testing environment. This framework will (1) allow manufacturers to easily and rapidly reconfigure and re-task robot systems in assembly operations, (2) make robots more accessible to small and medium organizations, (3) provide large organizations greater efficiency in their assembly operations, and (4) allow the U.S. to compete effectively in the global market.

In order to maintain compatibility with the IEEE working group, the ontology has been fully defined in the Web Ontology Language (OWL) [1]. In addition, the ontology is fully defined in the XML schema definition language (XSDL) [2]. Tools have been built in the APRS project that produce OWL class files automatically from XML schema files and produce OWL instance files conforming to OWL classes automatically from XML instance files conforming to the XML schema from which the OWL classes were built. This enables working in an XML environment and then producing OWL as needed. OWL and XSDL are largely but not fully compatible. The incompatibilities and how they are handled are discussed in section 6 of this paper.

The kitting workstation model is defined in OWL because the IEEE RAS Ontologies for Robotics and Automation Working Group has decided to use OWL, and the authors are participating in the activities of that working group. OWL allows the use of several different syntaxes. The functional-style syntax (which is the most compact one) has been used to write the OWL version of the kitting workstation model.

The OWL kitting workstation model is used by the OWL kitting plan model as well as for governing OWL kitting workstation instance files. Separate, slightly different versions of the model are used for those two purposes. None of those differences affects this description of the model.

This document uses the terms *class* and *data member* in their sense from common object-oriented model terminology. A class is a type of thing and has data members. Each data member has a type. For example, circle is a class, and diameter is a data member of that class that is a number.

# KITTING WORKSTATION ONTOLOGY OVERVIEW

As shown in Table 1 below, the model has two top-level classes, **SolidObject** and **DataThing**, from which all other classes are derived. **SolidObject** models solid objects, things made of matter. **DataThing** models data. The level of indentation in the table indicates subclassing. For example, **KitTray** is derived from **SkuObject**, and **SkuObject** is derived from **SolidObject**. Items in *italics* following classes are names of data members of the class. Derived types inherit the data members of the parent. Each data member has a specific type not shown in the table. If the type of a data member has derived types, any of the derived types may be used.

The names of the OWL properties that give the data members shown in the table are formed from the data member name by adding the prefix *has*class*\_* where class is the class name. For example, the name of the ObjectProperty for the *SolidObjects* data member of a **WorkTable** is *hasWorkTable\_SolidObjects*. For the XML representation, the prefixes are unnecessary and are not utilized.

Inverse properties are defined in the OWL version of the kitting workstation ontology for all of the ObjectProperties. The names of the inverse object properties are formed by changing the *has* at the beginning of the name to *hadBy* and reversing the order of the other two components of the name. For example, the inverse of *hasKit\_KitTray* is *hadByKitTray\_Kit*. The fact that two ObjectProperties are inverses is indicated by putting an InverseObjectProperties statement in the ontology.

In the table, (0) means zero or one of the data member may appear in an instance file, [0] means zero to many may appear, and [ ] means one to many may appear.

Table 1: Kitting Workstation Class Hierarchy

|  |
| --- |
| **SolidObject** *PrimaryLocation SecondaryLocation[0]* |
| **NoSkuObject** *InternalShape(0) ExternalShape(0)* |
| **EndEffector** *Description Weight MaximumLoadWeight HeldObject(0)* |
| **GripperEffector** |
| **VacuumEffector** *CupDiameter Length* |
| **VacuumEffectorMultiCup** *ArrayNumber ArrayRadius* |
| **VacuumEffectorSingleCup** |
| **EndEffectorChangingStation** *Base EndEffectorHolder[ ]* |
| **EndEffectorHolder** *EndEffector(0)* |
| **Human** |
| **Kit**  *DesignName KitTray Part[0] Slot[0] Finished* |
| **KittingWorkstation** *AngleUnit ChangingStation KitDesign[ ] LengthUnit  OtherObstacle[0] Robot Sku[ ] WeightUnit* |
| **LargeBoxWithEmptyKitTrays**  *LargeContainer KitTray[0]* |
| **LargeBoxWithKits** *LargeContainer Kit[0] KitDesign Capacity* |
| **MechanicalComponent** |
| **Robot** *Description EndEffector(0) MaximumLoadWeight WorkVolume* |
| **WorkTable** *ObjectOnTable[0]* |
| **SkuObject** *Sku* |
| **KitTray** *SerialNumber* |
| **LargeContainer**  *SerialNumber* |
| **Part**  *SerialNumber* |
| **PartsVessel** *SerialNumber PartSku PartQuantity Part[0]* |
| **PartsBin** |
| **PartsTray** |
|  |
| **DataThing** |
| **BoxVolume** *MaximumPoint MinimumPoint* |
| **KitDesign** *KitTraySku PartRefAndPose[ ]* |
| **PartRefAndPose** *Sku Point XAxis ZAxis* |
| **PhysicalLocation** *RefObject Timestamp(0)* |
| **PoseLocation** *Point ZAxis XAxis*  *PositionStandardDeviation(0) OrientationStandardDeviation(0)* |
| **PoseLocationIn** |
| **PoseLocationOn** |
| **PoseOnlyLocation** |
| **RelativeLocation** *Description* |
| **RelativeLocationIn** |
| **RelativeLocationOn** |
| **Point** *X Y Z* |
| **ShapeDesign** *Description GraspPose* |
| **ExternalShape** *ModelFormatName ModelFileName ModelName(0)* |
| **InternalShape** |
| **BoxyShape** *Length Width Height HasTop* |
| **CylindricalShape** *Diameter Height HasTop* |
| **Slot** *PartRefAndPose Part* |
| **StockKeepingUnit** *Description InternalShape(0) ExternalShape(0)*  *Weight EndEffector[0]* |
| **Vector** *I J K* |

Each **SolidObject** has a native coordinate system conceptually fixed to the object. The native coordinate system of a **SolidObject** with a **BoxyShape**, for example, has its origin at the middle of the bottom of the object, its Z axis perpendicular to the bottom, and the X axis parallel to the longer horizontal edges of the object.

Each **SolidObject** *A* has at least one *PhysicalLocation* (the *PrimaryLocation*). A *PhysicalLocation* is defined by giving a reference **SolidObject** *B* and information saying how the position of *A* is related to *B*. Two types of location are required for the operation of the kitting workstation. Relative locations, specifically the knowledge that one **SolidObject** is in or on another, are needed to support making logical plans for building kits. Mathematically precise locations are needed to support robot motion. The mathematical location, *PoseLocation*, gives the pose of the coordinate system of *A* in the coordinate system of *B*. The mathematical information consists of the location of the origin of *A*’s coordinate system and the directions of its Z and X axes. The mathematical location variety has subclasses representing that, in addition, *A* is in *B* (*PoseLocationIn*) or on *B* (*PoseLocationOn*). The subclasses of **RelativeLocation** are needed not only for logical planning, but also for cases when the relative location is known, but the mathematical information is not available. This occurs, for example when a **PartsBin** is being used, since by definition, the **Parts** in a **PartsBin** are located randomly.

All chains of location from **SolidObjects** to reference **SolidObjects** must end at a **KittingWorkstation** (which is the only class of **SolidObject** allowed to be located relative to itself).

For planning, it is assumed that **SolidObjects** do not move unless a command moves them. Also, if **SolidObject** *A* is in or on **SolidObject** *B* (so that the reference object for *A* is *B*), then if *B* is moved, the position of *A* relative to *B* is unchanged.

A **SolidObject** may be given multiple locations by using its *SecondaryLocation* data member. If multiple locations are used, they are expected to be logically and mathematically consistent.

The kitting ontology includes several subclasses of **SolidObject** that are formed from components that are **SolidObjects**. These are: **Kit**, **LargeBoxWithEmptyKitTrays**, and **LargeBoxWithKits**. Combined objects may come into existence or go out of existence dynamically when a kitting workstation is operating. For example, when all the kit trays in a **LargeBoxWithEmptyKitTrays** have been removed and put into kits, the **LargeBoxWithEmptyKitTrays** should go out of existence and the **LargeContainer** that was holding kit trays should have its location switched from its location relative to the **LargeBoxWithEmptyKitTrays** to the former location of the **LargeBoxWithEmptyKitTrays**.

The kitting workstation ontology provides two methods of describing shape: use a shape defined in the ontology (an **InternalShape**), or use a shape described in a separate file (an **ExternalShape**). Currently, there are only two types of **InternalShape**: **BoxyShape** and **CylindricalShape**. Details of these shapes are given in Section 5. An **ExternalShape** identifies the format and name of the file containing the shape. If that file contains more than one shape, the *ModelName* data member may be used to identify a particular one. Each **SolidObject** must have at least one of the two types of shape and may have both. If both types of shape are assigned, they should describe the same shape at different levels of detail.

In the kitting workstation ontology, there two principal subclasses of **SolidObject**: **SkuObject** and **NoSkuObject**. If the **SolidObject** is a **SkuObject**, it gets its shape from its SKU. If it is a **NoSkuObject**, it gets its shape directly.

# CLASS STRUCTURE DIAGRAMS

This section provides figures containing diagrams of the structure of the principal classes in the kitting workstation ontology.

All figures were generated by XMLSpy[[2]](#footnote-2) from the XML schema. In the figures, each box contains a data member of the class. A dotted line around a box means the data member is optional (may occur zero times). A *0..∞* underneath a box means it may occur zero or more times, with no upper limit on the number of occurrences. A *1..∞* underneath a box means it must occur at least once, with no upper limit on the number of occurrences. The irregular octagons with a line and three dots indicate sets of data members added at different levels of the inheritance hierarchy.

|  |
| --- |
| Figure 1: KittingWorkstation Model |

|  |
| --- |
| Figure 2: EndEffectorChangingStation Model |
| Figure 3: LargeBoxWithEmptyKitTrays Model |

|  |
| --- |
| Figure 4: LargeBoxWithKits Model |
| Figure 5: Kit Model |
| Figure 6: Robot Model |

The robot model is simple and does not currently have any kinematics or even any shape for the robot. It is likely that additional data members will be added in the future.

|  |
| --- |
| Figure 7: StockKeepingUnit Model |
| Figure 8: WorkTable Model    Figure 9: VacuumEffectorSingleCup Model    Figure 10: KitDesign Model    Figure 11: Part Model    Figure 12: PoseLocation Model DATATYPES The OWL language borrows most of the XSDL built-in datatypes. In addition, specialized OWL datatypes may be derived from the built-in datatypes by DatatypeDefinitions. These correspond to XSDL derivations of simple types by restricting built-in types. The OWL datatypes used in the kitting workstation ontology include:   * **angleUnit** – The **angleUnit** may be one of “degree” or “radian”. It specifies that any property that represents angles will be expressed in **angleUnit** units. This is defined by a DatatypeDefinition. * **boolean** – A **boolean** is the XSDL xsd:boolean. Valid values for a **boolean** are true, false, 0, and 1, where 0 is equivalent to false, and 1 equivalent to true. * **dateTime** – a **dateTime** is the XSDL xsd:dateTime. It represents the date and time in the form YYYY-MM-DDThh:mm:ss.sss, where YYYY is the year, MM the month, DD the day, T a literal T, hh the hour (00 to 23), mm the minutes, and ss.sss the seconds (from which the .sss may be removed or extended). * **decimal** – a **decimal** is the XSDL xsd:decimal. It represents a decimal number of arbitrary precision. The format of a **decimal** is a sequence of digits optionally preceded by a sign ("+" or "-") and optionally containing a period. The value may start or end with a period. If the fractional part is 0 then the period and trailing zeros may be omitted. Leading and trailing zeros are permitted, but they are not considered significant. For example, the decimal values 8.0 and 8.000 are considered equal. * **lengthUnit** – The **lengthUnit** may be one of “inch”, “meter”, or “millimeter”. It specifies that any property that represents lengths will be expressed in **lengthUnit** units. This is defined by a DatatypeDefinition. * **NMTOKEN** – The **NMTOKEN** is the XSDL xsd:NMTOKEN. It represents a single string token.  **NMTOKEN** values may consist of letters, digits, periods (.), hyphens (-), underscores (\_), and colons (:). They may start with any of these characters.  **NMTOKEN** does not preserve white space, so any leading or trailing whitespace will be removed. In addition, no whitespace may appear within the value itself. * **nonNegativeInteger** – A **nonNegativeInteger** is the XSDL xsd:nonNegativeInteger. This is defined as an arbitrarily large nonnegative integer. The digits may be optionally preceded by a plus (+) sign. Leading zeros are permitted, but decimal points are not. * **positiveDecimal** – A **positiveDecimal** is derived from the XSDL xsd:decimal and is restricted to be greater than zero. This is defined by a DatatypeDefinition. * **positiveInteger** – A **positiveInteger** is the XSDL xsd:positiveInteger. This is defined as an arbitrarily large positive integer. The digits may be optionally preceded by a plus (+) sign. Leading zeros are permitted, but decimal points are not. * **string** – A **string** is the XSDL xsd:string. The type represents a character string that may contain any Unicode character allowed by OWL. The **string** type preserves white space, which means that all whitespace characters (spaces, tabs, carriage returns, and line feeds) are preserved. * **weightUnit –** The **weightUnit** may be one of “gram”, “kilogram”, “milligram” “ounce”, or “pound”. It specifies that any property that represents weights will be expressed in **weightUnit** units. This is defined by a DatatypeDefinition. |

# Kitting WORKSTATION ontology details

This section provides descriptions of each of the classes in the kitting workstation ontology, in alphabetical order. The names used below are those in the OWL version of the ontology. Each description lists the data members of the class and explains what the class and its data members mean. The terms *inherited*, *optional*, and *multiple* that follow many data member names mean the obvious things. The term *inverse* means that the property (*hasThis\_That*) for the data member has an inverse. This occurs whenever the value of a data member is an object rather than a datatype and the data member is not inherited. In this case a *hadByThat\_This* property for the data member is also defined in the OWL version of the ontology. If the data member is inherited, some ancestor of the class has the inverse.

**BoxVolume** is derived from DataThing.

An instance of BoxVolume has the following:

MaximumPoint (inverse)

MinimumPoint (inverse).

The MaximumPoint and MinimumPoint are diagonally opposite corner points of a box shaped volume whose edges are aligned with the coordinate system in which the BoxVolume is located. The MinimumPoint has the minimum values of X, Y, and Z. The MaximumPoint has the maximum values of X, Y, and Z.

**BoxyShape** is derived from InternalShape.

An instance of BoxyShape has the following:

Description (inherited)

GraspPose (inherited, optional)

Length

Width

Height

HasTop.

A BoxyShape is box shaped. The Length is larger of the two dimensions that are not the Height. The Width is smaller of the two dimensions that are not the Height. The coordinate system of a BoxyShape (i.e. the thing that is located and oriented by a Pose) has its origin in the middle of the bottom, its Z-axis parallel to the height sides and pointing into the box, and its X-axis parallel to the length sides. If HasTop is true, the top of the box (i.e. the side through which the +Z axis passes) exists and is closed. If HasTop is false, the box has no top.

**CylindricalShape** is derived from InternalShape.

An instance of CylindricalShape has the following:

Description (inherited)

GraspPose (inherited, optional)

Diameter

Height

HasTop.

The cylinder is a right circular cylinder with a circular base having the given Diameter. The axis is perpendicular to the base. The base is always a surface that is part of the cylinder. The sides of the cylinder stop at the given Height as if cut by a plane perpendicular to the axis. The coordinate system of a CylindricalShape (i.e. the thing that is located and oriented by a pose) has its origin in the middle of the bottom, and its Z-axis on the axis of the cylinder. If HasTop is true, the top of the cylinder (i.e. the side through which the +Z axis passes) exists and is closed. If HasTop is false, the cylinder has no top.

**DataThing** is an abstract type from which more specific types of data thing are derived. That includes all complex data types such as Vector, PhysicalLocation, etc.

**EndEffector** is derived from NoSkuObject.

An instance of EndEffector has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

Description

Weight

MaximumLoadWeight

HeldObject (optional, inverse).

EndEffector is an abstract type from which more specific types of end effector are derived. An EndEffector is an end effector for a robot. The optional HeldObject is for the object being held by the end effector, if the end effector is holding an object. Every EndEffector is either a GripperEffector or a VacuumEffector. Every EndEffector in a KittingWorkstation is either attached to the end of a robot arm or sitting in an EndEffectorHolder at an EndEffectorChangingStation.

**EndEffectorChangingStation** is derived from NoSkuObject.

An instance of EndEffectorChangingStation has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

Base (inverse)

EndEffectorHolder (multiple, inverse).

An EndEffectorChangingStation is a place where end effectors are stored and where the robot can change end effectors. The coordinate system of an EndEffectorChangingStation is in the same place as the coordinate system of its Base. The shape of an EndEffectorChangingStation may also be found from the shapes of the Base and the EndEffectorHolders and their relative positions.

**EndEffectorHolder** is derived from NoSkuObject.

An instance of EndEffectorHolder has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

EndEffector (optional, inverse).

An EndEffectorHolder holds zero or one end effector and is part of an EndEffectorChangingStation.

**ExternalShape** is derived from ShapeDesign.

An instance of ExternalShape has the following:

Description (inherited)

GraspPose (inherited, optional)

ModelFormatName

ModelFileName

ModelName (optional).

An ExternalShape is a shape defined in an external file. The ModelFormatName is the name of the format of model (for example, 'STEP Advanced Brep' or 'USARSim'). The ModelFileName is the name of the file containing the model and may include a path (for example 'partFiles/STEP/ANC101.stp'). The model file may contain more than one shape model. The ModelName is optional and is the name of a model within the model file. The ModelName is necessary if the model file contains more than one model.

**GripperEffector** is derived from EndEffector.

An instance of GripperEffector has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

Description (inherited)

Weight (inherited)

MaximumLoadWeight (inherited)

HeldObject (inherited, optional).

A GripperEffector holds an object by gripping it with fingers or claws or by suction.

**Human** is derived from NoSkuObject.

An instance of Human has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional).

A Human is a type representing a human being. An internal shape used with a human should be a bounding box or cylinder that encloses the human completely.

**InternalShape** is derived from ShapeDesign.

An instance of InternalShape has the following:

Description (inherited)

GraspPose (inherited, optional).

InternalShape is an abstract type from which more specific types of shape are derived. Instances of InternalShape in a instance file contain information about the appearance of the shape without referring to another file.

**Kit** is derived from NoSkuObject.

An instance of Kit has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

Design (inverse)

KitTray (inverse)

Finished

Part (optional, multiple, inverse)

Slot (optional, multiple).

Finished is a boolean indicator of whether the Kit is finished. Part may occur several times (once for each part in the kit). The optional Slots may be used to keep track of whether each place in the kit that should have a part on it does have a part on it. The PartRefAndPose of each Slot should indicate a PartRefAndPose in the design of the kit (different for each slot). The locating point of the Tray in the kit should be (0,0,0), and its X and Z axes should be (1,0,0) and (0,0,1), respectively.

**KitDesign** is derived from DataThing.

An instance of KitDesign has the following:

KitTraySku (inverse)

PartRefAndPose (multiple, inverse).

The KitTraySku identifies a type of kit tray. The Pose in a PartRefAndPose is the location of the part relative to the coordinate system of the ShapeDesign of the tray.

**KittingWorkstation** is derived from NoSkuObject.

An instance of KittingWorkstation has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

AngleUnit

LengthUnit

ChangingStation (inverse)

Object (multiple, inverse)

OtherObstacle (optional, multiple, inverse)

Robot (inverse)

KitDesign (multiple, inverse)

Sku (multiple, inverse)

WeightUnit.

All angle, length, and weight values related to the workstation use the units implicitly. The workstation includes one robot and one end effector changing station. There may be many instances of Object in the workstation, including such things as work tables, large boxes with kits, large boxes with empty kit trays, and parts trays. The collection of instances of KitDesign is a library of all kit designs known to the workstation. The collection of instances of Sku is a library of all stock keeping units known to the workstation. The OtherObstacles are obstacles to robot motion of unspecified type. Containers of various sorts enter and leave the workstation. The robot builds kits of parts by executing kitting plans as directed by a kitting plan execution system. The location of each instance of KittingWorkstation should be given relative to itself in order to end the chain of relative locations.

**KitTray** is derived from SkuObject.

An instance of KitTray has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

Sku (inherited)

SerialNumber.

The Sku specifies the SKU of the kit tray. A KitTray is designed to hold Parts with various SKUs in known positions.

**LargeBoxWithEmptyKitTrays** is derived from NoSkuObject.

An instance of LargeBoxWithEmptyKitTrays has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

LargeContainer (inverse)

KitTray (optional, multiple, inverse).

The location point of the LargeContainer should be (0,0,0), its Z axis should be (0,0,1), and its X axis should be (1,0,0). The PrimaryLocation of a KitTray in a LargeBoxWithEmptyKitTrays should be given by a PoseLocationIn or RelativeLocationIn that is relative to the LargeContainer. The KitTrays in a LargeBoxWithEmptyKitTrays are intended to all be of the same SKU, although there is currently no formal requirement for that.

**LargeBoxWithKits** is derived from NoSkuObject.

An instance of LargeBoxWithKits has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

LargeContainer (inverse)

Kit (optional, multiple, inverse)

KitDesign (inverse)

Capacity.

The coordinate system of a LargeBoxWithKits is in the same place as the coordinate system of its LargeContainer. The PrimaryLocation of the LargeContainer should be relative to the LargeBoxWithKits. The KitDesign is an identifier for a KitDesign. The PrimaryLocation of a Kit in a LargeBoxWithKits should be given by a PoseLocationIn or RelativeLocationIn that is relative to the LargeContainer. The Capacity is an xs:positiveInteger giving the maximum number of kits of the given design that can be held in the box. The Kits in a LargeBoxWithKits are intended to all be of the given design, but there is currently no formal constraint requiring that.

**LargeContainer** is derived from SkuObject.

An instance of LargeContainer has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

Sku (inherited)

SerialNumber.

The Sku specifies the SKU of the LargeContainer. A LargeContainer can hold one or more instances of a single type of tray, bin, or kit.

**MechanicalComponent** is derived from NoSkuObject.

An instance of MechanicalComponent has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional).

A MechanicalComponent is a component of kitting workstation device such as a robot or an end effector changing station.

**NoSkuObject** is derived from SolidObject.

An instance of NoSkuObject has the following:

PrimaryLocation (inherited)

SecondaryLocations (inherited, optional, multiple)

InternalShape (optional, inverse)

ExternalShape (optional, inverse).

A NoSkuObject is an abstract type from which more specific types of solid object are derived. The InternalShape and ExternalShape are not required to represent the same shape, but they should not be inconsistent. If a NoSkuObject consists of components it may also get its shape from the shape of the components and their relative positions.

**Part** is derived from SkuObject.

An instance of Part has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

Sku (inherited)

SerialNumber.

The Part represents a part. The Sku specifies the SKU for the part.

**PartRefAndPose** is derived from DataThing.

An instance of PartRefAndPose has the following:

Sku (inverse)

Point (inverse)

ZAxis (inverse)

XAxis (inverse).

The Sku identifies a type of part. The Point specifies the location of the origin of the part in the coordinate system of the tray of the KitDesign to which the PartRefAndPose belongs. The ZAxis and XAxis specify the orientation of the part relative to that coordinate system.

**PartsBin** is derived from PartsVessel.

An instance of PartsBin has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

Sku (inherited)

SerialNumber (inherited)

PartSku (inherited)

PartQuantity (inherited)

Part (inherited, optional, multiple).

The Sku specifies the SKU for the PartsBin. A PartsBin holds a number of Parts (PartQuantity) with the same SKU (PartSku) in unknown random positions. Each Part in the tray should be listed explictly and have a RelativeLocationIn with the bin as its RefObject.

**PartsVessel** is derived from SkuObject.

An instance of PartsVessel has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

Sku (inherited)

SerialNumber

PartSku (inverse)

PartQuantity

Part (optional, multiple, inverse)

PartsVessel is an abstract type from which more specific types of things that supply parts are derived. The Sku specifies the SKU for the PartsVessel. The shape of a PartsVessel is as specified in its Sku. The PartSku specifies the SKU for the Parts in the PartsVessel. The value of PartQuantity should be the number of instances of Part.

**PartsTray** is derived from PartsVessel.

An instance of PartsTray has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

Sku (inherited)

SerialNumber (inherited)

PartSku (inherited)

PartQuantity (inherited)

Part (inherited, optional, multiple).

The Sku specifies the SKU of the PartsTray. A PartsTray holds a number of Parts (PartQuantity) with the same SKU (PartSku) in known positions. Each Part in the tray should be listed explictly and have a PoseLocation with the parts tray as its RefObject.

**PhysicalLocation** is derived from DataThing.

An instance of PhysicalLocation has the following:

RefObject (inverse)

Timestamp (optional).

PhysicalLocation is an abstract type from which more specific types of physical location are derived. A PhysicalLocation says where a SolidObject is relative to its reference object. Timestamp represents the most recent date and time when the location was updated.

**Point** is derived from DataThing.

An instance of Point has the following:

X

Y

Z.

X, Y, and Z are the Cartesian coordinates of the Point.

**PoseLocation** is derived from PhysicalLocation.

An instance of PoseLocation has the following:

RefObject (inherited)

Timestamp (inherited, optional).

Point (inverse)

XAxis (inverse)

ZAxis (inverse)

PositionStandardDeviation (optional)

OrientationStandardDeviation (optional).

PoseLocation is an abstract type from which more specific types of pose location are derived. The Point locates the origin of a coordinate system. The XAxis and ZAxis give the orientation of the coordinate system. The data for the Point, the ZAxis and the XAxis are expressed relative to the coordinate system of the reference object.

The PositionStandardDeviation is based on a normal distribution of actual position about its given value. Thus, for example, the actual position is expected to be within the given PositionStandardDeviation amount 68% of the time and within twice the given amount 95% of the time. The PositionStandardDeviation is measured in the length units being used.

The OrientationStandardDeviation is based on a normal distribution of orientation about its given value. The error is to be measured as the angle of rotation about a single axis needed to rotate a solid object from its stated orientation to its actual orientation. The OrientationStandardDeviation is measured in the angle units being used.

**PoseLocationIn** is derived from PoseLocation.

An instance of PoseLocationIn has the following:

RefObject (inherited)

Timestamp (inherited, optional).

Point (inherited)

XAxis (inherited)

ZAxis (inherited)

PositionStandardDeviation (inherited, optional)

OrientationStandardDeviation (inherited, optional).

A PoseLocationIn indicates that the object is inside the RefObject. The notion of 'inside' is vague and might be made more precise.

**PoseLocationOn** is derived from PoseLocation.

An instance of PoseLocationOn has the following:

RefObject (inherited)

Timestamp (inherited, optional).

Point (inherited)

XAxis (inherited)

ZAxis (inherited)

PositionStandardDeviation (inherited, optional)

OrientationStandardDeviation (inherited, optional).

A PoseLocationOn indicates that the Object is on top of the RefObject. The notion of 'on top of' is vague and might be made more precise.

**PoseOnlyLocation** is derived from PoseLocation.

An instance of PoseOnlyLocation has the following:

RefObject (inherited)

Timestamp (inherited, optional).

Point (inherited)

XAxis (inherited)

ZAxis (inherited)

PositionStandardDeviation (inherited, optional)

OrientationStandardDeviation (inherited, optional).

An object located by a PoseOnlyLocation may or may not be inside or on top of the reference object of the PoseOnlyLocation.

**RelativeLocation** is derived from PhysicalLocation.

An instance of RelativeLocation has a the following:

RefObject (inherited)

Timestamp (inherited, optional)

Description.

RelativeLocation is an abstract type from which more specific types of relative location are derived. A RelativeLocation indicates that the SolidObject that has the RelativeLocation is on or in the RefObject. The Description may be used to describe the relative positions of the object and its reference object.

**RelativeLocationIn** is derived from RelativeLocation.

An instance of RelativeLocationIn has the following:

RefObject (inherited)

Timestamp (inherited, optional)

Description (inherited).

A RelativeLocationIn indicates that the SolidObject that has the RelativeLocation is in the RefObject. The notion of 'in' is vague and might be made more precise.

**RelativeLocationOn** is derived from RelativeLocation.

An instance of RelativeLocationOn has the following:

RefObject (inherited)

Timestamp (inherited, optional)

Description (inherited).

A RelativeLocationOn indicates that the SolidObject that has the RelativeLocation is on top of the the RefObject. The notion of 'on top of' is vague and might be made more precise.

**Robot** is derived from NoSkuObject.

An instance of Robot has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

Description

EndEffector (optional, inverse)

MaximumLoadWeight

WorkVolume (multiple, inverse).

The Robot ontology given here might be expanded greatly to include, for example, its kinematic description, the values of joint angles, arm lengths of variable length arms, gripper actuation (open, closed, etc.), ranges, velocities, and accelerations of each joint, etc.

**ShapeDesign** is derived from DataThing.

An instance of ShapeDesign has the following:

Description (optional)

GraspPose (optional, inverse).

ShapeDesign is an abstract type from which more specific types of shape design are derived. Each ShapeDesign has a coordinate system that is expected to be specified explicitly or implicitly. A shape defined using coordinate values has an implicit coordinate system. The GraspPose is relative to the coordinate system of the ShapeDesign. The Point in the pose is the point at which a gripper should make contact with the shape. The ZAxis of the pose may be used to indicate a direction for aligning the ZAxis of the gripper (parallel or antiparallel) and is usually normal to the the object having the shape and pointing away from the object. The GraspPose should not use the optional Timestamp.

**SkuObject** is derived from SolidObject.

An instance of SkuObject has the following:

PrimaryLocation (inherited)

SecondaryLocations (inherited, optional, multiple)

Sku (inverse)

A SkuObject is an abstract type from which more specific types of solid object are derived. A SkuObject is an instance of a stockkeeping unit. The shape of a SkuObject is specified by its stockkeeping unit.

**Slot** is derived from DataThing.

An instance of Slot has the following:

PartRefAndPose (inverse)

Part (optional, inverse).

A Slot identifies whether or not a particular PartRefAndPose from the design of a Kit is occupied in an instance of a Kit. The PartRefAndPose identifies a PartRefAndPose from the Design of the Kit. The Part identifies a Part that occupies the PartRefAndPose. The Sku of the PartRefAndPose should be the Sku of the Part, the PartRefAndPose should be in the Kit design, and the Part should be in the Kit. The location described by the Pose of the Part in the Kit may differ from the location described by the Pose in the PartRefAndPose, but will usually be very close to it. If the Part is not used for a slot, that means the slot is empty.

**SolidObject** is an abstract type from which more specific types of solid thing are derived.

An instance of SolidObject has the following:

PrimaryLocation (inverse)

SecondaryLocations (optional, multiple, inverse)

The secondary locations are required to be logically and mathematically consistent with the value of the PrimaryLocation so that all locations of a SolidObject describe (or are consistent with) a single place in space. No SolidObject except the Workstation may be located with respect to itself, and all chains of primary location must end at the Workstation.

**StockKeepingUnit** is derived from DataThing.

An instance of StockKeepingUnit has the following:

Description

InternalShape (optional, inverse)

ExternalShape (optional, inverse)

Weight

EndEffector (optional, multiple, inverse).

A StockKeepingUnit is an object type description. SKU is an abbreviation for Stock Keeping Unit. Each EndEffector identifies an instance of EndEffector that can handle the SKU. One or both of InternalShape and ExternalShape must be given. The shapes are not required to represent the same shape, but they should not be inconsistent.

**VacuumEffector** is derived from EndEffector.

An instance of VacuumEffector has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

Description (inherited)

Weight (inherited)

MaximumLoadWeight (inherited)

CupDiameter

Length.

VacuumEffector is an abstract type from which more specific types of VacuumEffector are derived. A VacuumEffector holds an object by putting a cup or cups against the object and applying a vacuum.

**VacuumEffectorMultiCup** is derived from VacuumEffector.

An instance of VacuumEffectorMultiCup has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

Description (inherited)

Weight (inherited)

MaximumLoadWeight (inherited)

CupDiameter (inherited)

Length (inherited)

ArrayNumber

ArrayRadius.

The ArrayNumber is the number of cups, which must be at least 2. The cups are arranged in a circular array spaced evenly apart. The center of the wide end of one cup is on the X-axis of the coordinate system of the VacuumEffectorMultiCup. The center of the circular array is at the origin of the coordinate system. The axis of the array circle is the Z axis of the coordinate system, and the length of the VacuumEffector is measured along that axis. The wide ends of the cups lie on the XY plane of the coordinate system. Note that a square array can be represented easily as circular array.

**VacuumEffectorSingleCup** is derived from VacuumEffector.

An instance of KitTray has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

Description (inherited)

Weight (inherited)

MaximumLoadWeight (inherited)

CupDiameter (inherited)

Length (inherited).

A VacuumEffectorSingleCup has one cup. The center of the wide end of the cup (which is a circle) is at the origin of the coordinate system of the VacuumEffectorSingleCup. The Z axis of the coordinate system is the axis of that circle, and the length of the VacuumEffector is measured along that axis.

**Vector** is derived from DataThing.

An instance of Vector has the following:

I

J

K.

I, J, and K represent the usual i, j, and k components of a 3D vector.")

**WorkTable** is derived from NoSkuObject.

An instance of WorkTable has the following:

PrimaryLocation (inherited)

SecondaryLocation (inherited, optional, multiple)

InternalShape (inherited, optional)

ExternalShape (inherited, optional)

ObjectOnTable (optional, multiple, inverse).

Each ObjectOnTable is a SolidObject located with respect to the WorkTable. The reference object of each ObjectOnTable should be the WorkTable. Typically, those objects will be on top of the WorkTable. Typically, the shape of a WorkTable will be a BoxyShape, so that the table has Length, Width, and Height.

# Differences BetweeN OWL and XSDL

Although the OWL and XML schema kitting workstation models are conceptually identical, there are some systematic differences between the models caused by incompatibilities in the languages that go beyond the differences inherent in using two different languages.

* 1. XSDL Elements vs. OWL Properties

Conceptually, both versions of the model define classes, and the classes have data members. In OWL, classes are called *classes*, whereas in XSDL they are called *complexTypes*. It is standard practice in XSDL to have each complexType name end in “Type”. ComplexTypes have data members called *elements*, and each element must be of a particular type. It is convenient and helpful in XSDL to make the name of each element whose type is a complexType be identical to the name of its type, with the final “Type” removed. For example, the type of an element named **WorkTable** would be **WorkTableType**. In OWL, a data member is associated with a class by defining a *DataProperty* or an *ObjectProperty*, and having the data member be the value of the property. The property names are compound names formed as described earlier, in which the “Type” suffix is not needed. Hence, the complexType names (i.e. class names) in the XSDL version of the model have the “Type” suffix while the corresponding class names in the OWL version omit the “Type” suffix.

* 1. Name

All of the complexTypes in the XSDL version of the kitting workstation ontology have a “Name” element that is not present in the OWL version. A Name is not needed in OWL because names are assigned as a matter of course when instances of classes are created.

* 1. List of Objects

The XML schema model has a list of “Object” elements. This includes work tables and all of the movable solid objects located relative to the workstation. The OWL model does not have a corresponding list. In an OWL instance file, these solid objects may appear anywhere.

* 1. Multiple Inheritance

XSDL does not support multiple inheritance. OWL supports multiple inheritance, but that has not been used in the kitting workstation ontology. Except by subclass relationship, no object is in more than one class.

* 1. File Combination and Prefixes

Both XSDL and OWL enable combining separate models into a single model and use different prefixes to help avoid naming conflicts. In XSDL, several files may use the same prefix (or no prefix), but in OWL, each file must have its own prefix. The XSDL version of the kitting workstation ontology uses no prefix. The OWL version of the ontology uses the prefix ktw: (a very short abbreviation of “kitting workstation”). The ktw: prefix is not shown in this document, but it appears in the formal OWL version of the kitting workstation ontology. The empty OWL prefix “:” has been reserved for OWL instance files. In OWL, instance files are combined with class files, so the class files must have prefixes.

# OWL CLASS FILE ORGANIZATION

As mentioned earlier, the OWL kittingWorkstation class file is generated automatically from the XSDL version of the model. Hence, it is very regularly organized. The overall outline of the file is shown below. While few people will want to read the file as text, it is arranged neatly so that those who do read it will find the layout easy to follow. Each of the items listed below starts on a new line.

Header  
 Prefix declarations

Ontology declaration

Ontology comment (brief description of the ontology)

Prefix comment

Datatype definitions

Class definitions (each one has the following outline)

Class declaration

Class comment (description of the class)

Subclass declarations (if the class has subclasses)

Disjoint union declaration (if the class has subclasses)

Class properties (if there are any)

If a property is an object property, it has the following items (in the order shown). Some lines exist only if certain conditions hold, as shown in parentheses.

Object property declaration (of the form *hasThis\_That*)

Object property domain

Object property range

InverseFunctional (if each object in the range is the property value for only one object in the domain)

Functional object property (if there is zero or one object in the range for each object in the domain)

Equivalent classes (if every object in the domain has a value for the property)

*Blank line*

Object property declaration for the inverse (of the form *hadByThat\_This*)

Inverse properties declaration (*hasThis\_That* and *hadByThat\_This* are inverse properties)

Object property domain of the inverse

Object property range of the inverse

*Blank line*

HasKey declaration (in a very few cases)

If a property is a data property, it has the following items (in the order shown).

Data property declaration (of the form *hasThis\_That*)

Data property domain

Data property range

Functional data property (if there is zero or one object in the range for each object in the domain)

Equivalent classes (if every object in the domain has a value for the property)

# For more information

For more information about the industrial kitting ontology, please contact:

[craig.schlenoff@nist.gov](mailto:craig.schlenoff@nist.gov)

301-975-3456

References

[1] F. Harmelen and D. McGuiness, "OWL Web Ontology Language Overview, W3C web site: <http://www.w3.org/TR/2004/REC-owl-features-20040210/,>" 2012.

[2] P. Walmsley, *Definitive XML Schema*. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

1. http://lissi.fr/ora/doku.php [↑](#footnote-ref-1)
2. Certain commercial/open source software and tools are identified in this paper in order to explain our research. Such identification does not imply recommendation or endorsement by the authors, nor does it imply that the software tools identified are necessarily the best available for the purpose. [↑](#footnote-ref-2)