The Industrial Kitting Ontology Version 0.5

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

November 19, 2012

1. OVERVIEW

The purpose of this paper is to document the contents of the industrial kitting ontology being developed at the National Institute of Standards and Technology (NIST) as part of the Knowledge-Driven Robot Planning (KDRP) project[1] in the Next Generation Robotics and Automation program[2]. 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[3].

The goal of the KDRP project is to develop the measurement science and standards to enable advancements in the autonomous decision-making of robots through application to specific scenarios relevant to manufacturing. The ability to create and execute plans in real-world scenarios is what separates a machine-tool from an intelligent manufacturing system. Plans enable a robot to change its actions to deal with uncertainty in its environment and to rapidly switch to new tasks. These plans are based on models of the current environment, predictions about the future, and a priori knowledge of causal relationships between current actions and results. The breadth and usability of knowledge in these models is one of the main factors that constrains the flexibility and performance of manufacturing planning systems. Currently, however, there is no accepted standardized way to represent this knowledge, to reason with this knowledge, or to measure the performance of these systems. The goal of the industrial kitting ontology is to provide this standard knowledge representation.

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 was also fully defined in the XML schema language [2]. Although the two models are conceptually identical, there are some systematic differences between the models (in addition to differences inherent in using two different languages).

- The complexType names (i.e. class names) in XML schema have the suffix "Type" added which is not used in OWL. This is so that the same names without the suffix can be used in XML schema language as element names without confusion.
- All of the XML schema complexTypes have a "Name" element that is not present in OWL. It is not needed in OWL because names are assigned as a matter of course when instances of classes are created.
- The XML schema model has a list of "Object" elements. This collects all of the movable objects. The OWL model does not have a corresponding list. In an OWL data file, the movable objects may appear anywhere.

[1] http://www.nist.gov/el/isd/ps/intellplanmodautosys.cfm
[2] http://www.nist.gov/el/isd/ps/nextgenrobauto.cfm
[3] http://lissi.fr/ora/doku.php

- OWL has classes but does not have attributes; it has ObjectProperties and DataProperties instead. They may be used to model attributes. OWL Properties are global, not local to a class, so localizing each attribute to a class is done by a naming convention that includes using prefixes as described below. The prefixes are not used in XML schema.
- OWL supports multiple inheritance, but that has not been used in the kitting ontology. Except by subclass relationship, no object is in more than one class.

The kitting workstation model was defined first 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 model is grounded through the use of DatatypeDefinitions in OWL (simple types in XML schema as seen in http://www.schemacentral.com. All of the classes may be traced down to these basic types. The types include:

- **angleUnit** – The **angleUnit** is constrained from the XML type xsd:NMTOKEN and may be one of “degree” or “radian”. It specifies that any property that represents angles will be expressed in **angleUnit** units.
- **boolean** – A **boolean** is of the XML type xsd:boolean. Valid values for a **boolean** are true, false, 0, and 1.
- **decimal** – a **decimal** is of the XML type 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** is constrained from the XML type xsd:NMTOKEN and may be one of “inch”, “meter”, or “millimeter”. It specifies that any property that represents lengths will be expressed in **lengthUnit** units.
- **NMTOKEN** – The **NMTOKEN** is of the XML type 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 of the XML type 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 constrained from the XML type xsd:decimal and is defined as a positive **decimal** greater than zero.
- **positiveInteger** – A **positiveInteger** is of the XML type 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 of the XML type 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** is constrained from the XML type xsd:NMTOKEN and may be one of "gram", "kilogram", "milligram" "ounce", or "pound". It specifies that any property that represents weights will be expressed in **weightUnit** units.

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. Subclasses of **SolidObject** and **DataThing** are defined as shown in Table 1. The level of indentation indicates subclassing. For example, **WorkTable** is derived from **BoxyObject**, and **BoxyObject** is derived from **SolidObject**. Items in *italics* following classes are names of class attributes. Derived types inherit the attributes of the parent. Each attribute has a specific type not shown in the listing below. If an attribute type has derived types, any of the derived types may be used.

The names of the OWL properties that give the attributes shown in the table above are formed from the attribute name by adding the prefix *has*class_ where class is the class name. For example, the name of the ObjectProperty for the *SolidObjects* attribute of a **WorkTable** is *hasWorkTable_SolidObjects*. For the XML representation, the prefixes are unnecessary and are not utilized.

Many of the ObjectProperties in the kitting ontology have inverses. 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_Parts* is *hadByPart_Kit*. The fact that two ObjectProperties are inverses is indicated by putting an InverseObjectProperties statement in the ontology.

Each **SolidObject** has a native coordinate system conceptually fixed to the object. The native coordinate system of a **BoxyObject**, 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.

Table 1: Kitting Object Ontology Overview

SolidObject *PrimaryLocation SecondaryLocation*
BoxyObject *Length Width Height*
WorkTable *SolidObjects*
EndEffector *Description Weight Id LoadWeight*
GripperEffector
VacuumEffector *CupDiameter Length*
VacuumEffectorMultiCup *ArrayNumber ArrayRadius*

VacuumEffectorSingleCup
EndEffectorChangingStation *EndEffectorHolders*
EndEffectorHolder *EndEffector*
Kit *Tray DesignRef Parts Finished?*
KittingWorkstation *WorkTable Robot ChangingStation AngleUnit LengthUnit WeightUnit KitDesigns OtherObstacles Skus*
KitTray *SkuRef SerialNumber*
LargeBoxWithEmptyKitTrays *LargeContainer Trays*
LargeBoxWithKits *LargeContainer Kits KitDesignRef Capacity*
LargeContainer *SkuRef SerialNumber*
Part *SkuRef SerialNumber*
PartsBin *PartQuantity PartSkuRef SkuRef SerialNumber*
PartsTray *SkuRef SerialNumber*
PartsTrayWithParts *PartTray*
Robot *Description MaximumLoadWeight EndEffector WorkVolume*
DataThing
BoxVolume *MaximumPoint MinimumPoint*
KitDesign *KItTraySkuRef PartRefAndPoses*
PartRefAndPose *SkuRef Point XAxis ZAxis*
PhysicalLocation *RefObject*
PoseLocation *Point ZAxis XAxis*
PoseLocationIn
PoseLocationOn
PoseOnlyLocation
RelativeLocation *Description*
RelativeLocationIn
RelativeLocationOn
Point *X Y Z*
ShapeDesign *Description*
BoxyShape *Length Width Height HasTop*
StockKeepingUnit *Description Shape Weight EndEffectorRefs*
Vector *I J K*

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* attribute. 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**, **PartsTrayWithParts**, **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 parts in a **PartsTrayWithParts** have been removed and put into kits, the **PartsTrayWithParts** should go out of existence and the **PartsTray** that was holding Parts should have its location switched from its location relative to the **PartsTrayWithParts** to the former location of the **PartsTrayWithParts**.

In the current version of the kitting ontology, there are two ways in which the shape of a **SolidObject** can be specified. First, if the **SolidObject** is a **BoxyObject**, it is shaped like a box with length, width, and height. A **WorkTable** gets its shape that way. Second, if the **SolidObject** has a *SkuRef*, its shape is the shape specified in the referenced **StockKeepingUnit**. That shape is given by a **ShapeDesign**, which is an abstract class that does not have any attributes that describe shape. Currently, there is one derived class of **ShapeDesign**, named **BoxyShape**. A **BoxyShape** is a box with *Length*, *Width*, and *Height*. A **BoxyShape** may or may not have a top, as specified by the boolean *HasTop* attribute. These two methods of describing shape are adequate for making kits of boxes, but are not adequate for most industrial forms of kitting. For manipulating non-boxy **SolidObjects** some mathematically usable representation of shape will be required. The most commonly used type of shape representation is the boundary representation. Boundary representations are too complex to be modeled in OWL, so either some interface from OWL to a boundary representation will need to be constructed, or a simpler but usable shape representation will need to be defined and modeled in OWL.

2. KITTING ONTOLOGY DETAILS

In this section we will take a deeper look at the structure of each portion of the ontology. All of the objects listed below are types of **SolidObjects**, which is an abstract class not intended to be instantiated.

The main portions of the ontology include:

- Kitting Workstation (Figure 1)
- Changing Station (Figure 2)
- Large Box With Empty Kit Trays (Figure 3)
- Large Box With Kits (Figure 4)
- Parts Tray With Parts (Figure 5)
- Robot (Figure 6)
- Stock Keeping Unit (SKU) (Figure 7)
- Work Table (Figure 8)

The structure of the **KittingWorkstation** class is shown in Figure 1. The figure shows the names of the attributes of a **KittingWorkstation**. The first two attributes (*PrimaryLocation* and *SecondaryLocation*) are inherited from the **SolidObject** class. The rest of the attributes are specific to the **KittingWorkstation** class. The *AngleUnit*, *LengthUnit*, and *WeightUnit* apply to all quantities in a data file that are in terms of those unit types. No other unit types are used in the model.

In Figures 1-8 (which were generated by XMLSpy[4] from an XML schema depicting the OWL model), a dotted line around a box means the attribute is optional (may occur zero times), while a *0..∞* underneath a box means it may occur more than once, with no upper limit on the number of occurrences.

[4] 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.

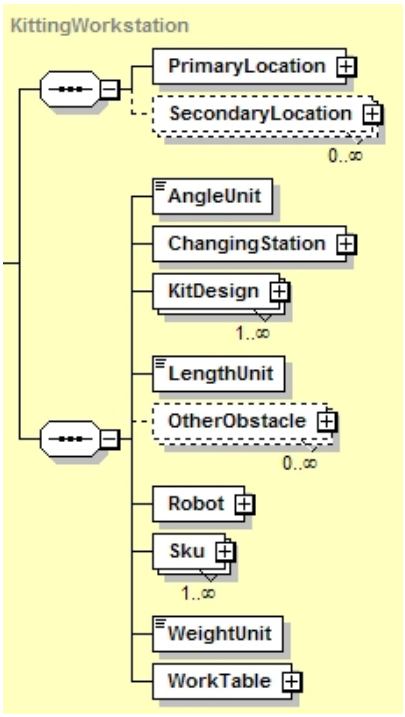

Figure 1: Kitting Workstation Model

Figure 2 through Figure 8 follow the same structure as what is shown in Figure 1.

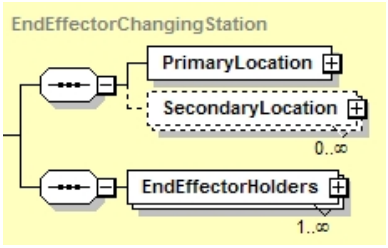

Figure 2: Changing Station Model

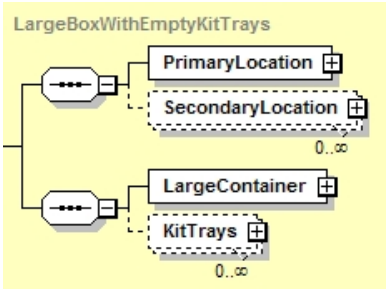

Figure 3: Large Box With Empty Kit Trays Model

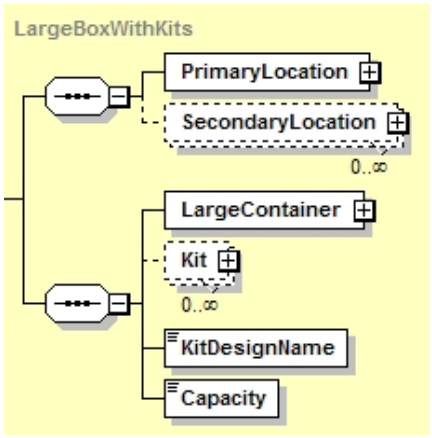

Figure 4: Large Box With Kits Model

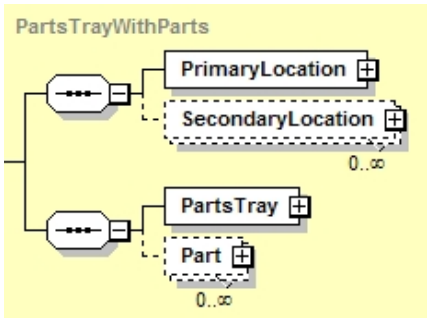

Figure 5: Parts Tray With Parts Model

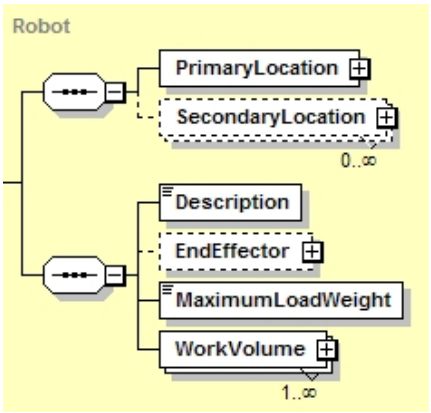

Figure 6: Robot Model

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

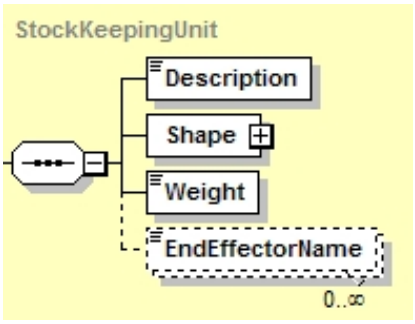

Figure 7: Stock Keeping Unit Model

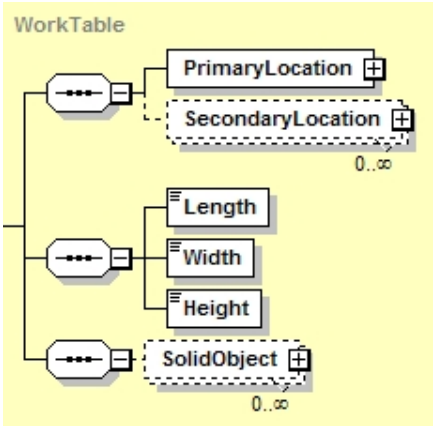

Figure 8: Work Table Model

3. KITTING ONTOLOGY DETAILS

In this section, we provide paragraph descriptions of each of the classes in the kitting ontology, in alphabetical order. The naming convention utilized below follows the OWL implementation of the ontology.

- **Box Volume** - A **BoxVolume** is a **DataThing**. A **BoxVolume** has a maximum **Point** (*hasBoxVolume_MaximumPoint*) and a minimum **Point** (*hasBoxVolume_MinimumPoint*). These 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 minimum **Point** has the minimum values of X, Y, and Z. The maximum **Point** has the maximum values of X, Y, and Z.

- **Boxy Object** - A **BoxyObject** is a **SolidObject**. A **BoxyObject** is box shaped. It has length, width, and height (*hasBox_Length*, *hasBox_Width*, *hasBox_Height*) of type **positiveDecimal**. It has a preferred partial orientation in which the edges along which the height is measured are vertical (parallel to the force of gravity). The length is the

larger of the two dimensions that are not the height. The width is the smaller of the two dimensions that are not the height. The coordinate system of a **BoxyObject** (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 its X-axis parallel to the larger length sides. Since this still allows two choices for orientation (four if the length and width are equal) which may or may not be distinguishable, some subtypes of **BoxyObject** will need one more piece of orientation information.

- **Boxy Shape** - A **BoxyShape** is a **ShapeDesign**. A **BoxyShape** is box shaped. It has length, width, and height (*hasBoxyShape_Length*, *hasBoxyShape_Width*, *hasBoxyShape_Height*) which are of type **positiveDecimal**, and a **boolean** indicator of whether it has a top, (*hasBoxyShape_hasTop*). It has a preferred partial orientation in which the edges along which the height is measured are vertical (parallel to the force of gravity). The length is the larger of the two dimensions that are not the height. The width is the 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 its X-axis parallel to the length sides.

- **Data Thing** - A **DataThing** is a **Thing**. A **DataThing** includes all complex data types such as **Vector**, **PhysicalLocation**, etc. Currently, it has no properties.

- **End Effector** - An **EndEffector** is a **SolidObject**. It is an end effector for a robot. An **EndEffector** has a description of type **string** (*hasEndEffector_Description*), a weight of type **positiveDecimal** (*hasEndEffector_Weight*), and a maximum weight it can lift of type **positiveDecimal** (*hasEffector_MaximumLoadWeight*). Every **EndEffector** is either a **GripperEffector** or a **VacuumEffector**. Every **EndEffector** in a **KittingWorkstation** is either attached to the end of a robot arm (*hadByEndEffector_Robot*) or sitting in an **EndEffectorHolder** (*hadByEndEffector_EndEffectorHolder*) at an **EndEffectorChangingStation**. An **EndEffector** must not be holding anything when it is placed in an **EndEffectorHolder** or when it is sitting in an **EndEffectorHolder**.

- **End Effector Changing Station** - An **EndEffectorChangingStation** is a **SolidObject**. It is a place where end effectors are stored and where the robot can change end effectors. It has **EndEffectorHolders** (*hasChangingStation_EndEffectorHolders*). An **EndEffectorChangingStation** belongs to a **KittingWorkstation** (*hadByChangingStation_Workstation*).

- **End Effector Holder** - An **EndEffectorHolder** is a **SolidObject**. An **EndEffectorHolder** holds zero or one **EndEffector** (*hasEndEffectorHolder_EndEffector*). An **EndEffectorHolder** is part of an **EndEffectorChangingStation** (*hadByEndEffectorHolder_ChangingStation*).

- **Gripper Effector** - A **GripperEffector** is an **EndEffector**. A **GripperEffector** holds an object by gripping it with fingers or claws.

- **Kit** - A **Kit** is a **SolidObject**. A **Kit** has a **KitDesign** (*hasKit_Design*), a **KitTray** (*hasKit_Tray*), a set of **Parts** (*hasKit_Parts*), and a **boolean** indicator of whether the **Kit** is finished (*isKit_Finished*). The coordinate system of a **Kit** is in the same place as the coordinate system of its **KitTray**. The **PrimaryLocation** of a **Part** in a **Kit** should be given by a **PoseLocationIn** that is relative to the **KitTray**. A **Kit** may belong to a **LargeBoxWithKits** (*hadByKit_LargeBoxWithKits*).

- **Kit Design** - A **KitDesign** is a **DataThing**. A **KitDesign** identifies a type of tray (*hasKitDesign_KitTraySku*), and intended poses of **Parts** in finished **Kits** of this design (*hasKitDesign_PartRefAndPoses*). The pose (Point, zAxis, and xAxis) in a **PartRefAndPose** specifies the location of the part relative to the coordinate system of the **ShapeDesign** of the tray. Each **KitDesign** belongs to a **KittingWorkstation** (*hadByKitDesign_Workstation*).

- **Kit Tray** - A **KitTray** is a **BoxyObject**. A **KitTray** is designed to hold **Parts** with various **StockKeepingUnit** ids in known positions. A **KitTray** has a **StockKeepingUnit** (*hasKitTray_Sku*) and a **SerialNumber** (*hasKitTray_SerialNumber*). A **KitTray** may belong to a **Kit** (*hadByKitTray_Kit*) or to a **LargeBoxWithEmptyKitTrays** (*hadByKitTray_LargeBoxWithEmptyKitTrays*).

- **Kitting Workstation** - A **KittingWorkstation** is a **SolidObject**. A **KittingWorkstation** contains a **WorkTable** (*hasWorkstation_WorkTable*), a **Robot** (*hasWorkstation_Robot*), an **EndEffectorChangingStation** (*hasWorkstation_ChangingStation*), and other fixed obstacles of type **BoxVolume** (*hasWorkstation_OtherObstacles*) such as a computer. A **KittingWorkstation** has an **angleUnit** (*hasWorkstation_AngleUnit*), a **lengthUnit** (*hasWorkstation_LengthUnit*) and a **weightUnit** (*hasWorkstation_WeightUnit*). All angle, length, and weight values related to the workstation must use those units. A **KittingWorkstation** has **StockKeepingUnits** it knows about (*hasWorkstation_Skus*). A **KittingWorkstation** has **KitDesigns** it knows about (*hasWorkstation_KitDesigns*). In addition, 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.

- **Large Box With Empty Kit Trays** - A **LargeBoxWithEmptyKitTrays** is a **SolidObject**. A **LargeBoxWithEmptyKitTrays** has a **LargeContainer** (*hasLargeBoxWithEmptyKitTrays_LargeContainer*) and a set of **KitTrays** which should be empty (*hasLargeBoxWithEmptyKitTrays_KitTrays*). The coordinate system of a **LargeBoxWithEmptyKitTrays** is in the same place as the coordinate system of its **LargeContainer**. 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 **StockKeepingUnit**, although there is currently no formal requirement for that.

- **Large Box With Kits** - A **LargeBoxWithKits** is a **SolidObject**. A **LargeBoxWithKits** has a **LargeContainer** (*hasLargeBoxWithKits_LargeContainer*), a set of **Kits** (*hasLargeBoxWithKits_Kits*), a **KitDesign** (*hasLargeBoxWithKits_KitDesign*), and a **positiveInteger** giving the maximum number of kits of the given design that can be held in the box (*hasLargeBoxWithKits_Capacity*). The coordinate system of a **LargeBoxWithKits** is in the same place as the coordinate system of its **LargeContainer**. The **PrimaryLocation** of a **Kit** in a **LargeBoxWithKits** should be given by a **PoseLocationIn** or **RelativeLocationIn** that is relative to the **LargeContainer**. The **Kits** in a **LargeBoxWithKits** are intended to all be of the given design, but there is currently no formal constraint requiring that.

- **Large Container** - A **LargeContainer** is a **SolidObject**. A **LargeContainer** can hold one or more instances of a single type of tray or bin. The single type may be a (1) **KitTray**, (2) **PartsBin**, (3) **PartsTray**, or (4) **Kit**. A **LargeContainer** has a **StockKeepingUnit** (*hasLargeContainer_Sku*) and a **SerialNumber** (*hasLargeContainer_SerialNumber*). A **LargeContainer** may belong to a **LargeBoxWithEmptyKitTrays** (*hadByLargeContainer_LargeBoxWithEmptyKitTrays*) or to a **LargeBoxWithKits** (*hadByLargeContainer_LargeBoxWithKits*).

- **Part** - A **Part** is a **SolidObject**. It has a **StockKeepingUnit** (*hasPart_Sku*) and a **SerialNumber** (*hasPart_SerialNumber*). A Part may belong to a **Kit** (*hadByPart_Kit*) or to a **PartsTrayWithParts** (*hadByPart_PartsTrayWithParts*).

- **Part Reference And Pose** - A **PartRefAndPose** is a **DataThing**. A **PartRefAndPose** identifies a type of part by giving its **StockKeepingUnit** (*hasPartRefAndPose_Sku*), it specifies the location of the **Part** (*hasPartRefAndPose_Point*), and the orientation of the **Part** (*hasPartRefAndPose_XAxis* and *hasPartRefAndPose_ZAxis*). The pose is relative to the coordinate system of the **KitTray** identified in the **KitDesign**. A **PartRefAndPose** belongs to a **KitDesign** (*hadByPartRefAndPose_KitDesign*).

- **Parts Bin** - A **PartsBin** is a **SolidObject** used to hold **Parts** in unknown positions. A **PartsBin** has a **StockKeepingUnit** (*hasPartsBin_Sku*), a **SerialNumber** (*hasPartsBin_SerialNumber*), the name of the **StockKeepingUnit** for the parts it contains (*hasPartsBin_SkuRef*), and the number of parts it holds of type **nonNegativeInteger** (*hasPartsBin_PartQuantity*).

- **Parts Tray** - A **PartsTray** is a **SolidObject** used to hold **Parts**. A **PartsTray** has a **StockKeepingUnit** (*hasPartsTray_Sku*) and a **SerialNumber** (*hasPartsTray_SerialNumber*). A **PartsTray** may belong to a **PartsTrayWithParts** (*hadByPartsTray_PartsTrayWithParts*).

- **Parts Tray With Parts** - A **PartsTrayWithParts** is a **SolidObject**. A **PartsTrayWithParts** has a **PartsTray** (*hasPartsTrayWithParts_Tray*) and a set of **Parts** (*hasPartsTrayWithParts_Parts*). The coordinate system of a **PartsTrayWithParts** is in the same place as the coordinate system of its **PartsTray**. The **PrimaryLocation** of a **Part** in a **PartsTrayWithParts** should be given by a **PoseLocationIn** that is relative to the **PartsTray**. The **Parts** in a **PartsTrayWithParts** are intended to all be of the same **StockKeepingUnit**, although there is currently no formal requirement for that.

- **Pose Location** - A **PoseLocation** is a **PhysicalLocation**. A **PoseLocation** consists of a **Point** (*hasPoseLocation_Point*), a **Vector** for the Z axis (*hasPoseLocation_ZAxis*), a **Vector** for the X axis (*hasPoseLocation_XAxis*), and a reference object inherited from the **PhysicalLocation** class (*hasPhysicalLocation_RefObject*). The data for the **Point**, the Z axis and the X axis are expressed relative to the coordinate system of the reference object. A **PoseLocation** must be a **PoseOnlyLocation**, a **PoseLocationIn**, or a **PoseLocationOn**.

- **Pose Only Location** - A **PoseOnlyLocation** is a **PoseLocation**. A **PoseOnlyLocation** consists of a **Point** (*hasPoseLocation_Point*), a **Vector** for the Z axis (*hasPoseLocation_ZAxis*), a **Vector** for the X axis (*hasPoseLocation_XAxis*), and a reference object (*hasPhysicalLocation_RefObject*) all inherited from **PoseLocation**. The data for the **Point**, the Z axis and the X axis are expressed relative to the coordinate system of the reference object. An object located by a **PoseOnlyLocation** may or may not be inside or on top of the reference object of the **PoseOnlyLocation**.

- **Pose Location In** - A **PoseLocationIn** is a **PoseLocation**. A **PoseLocationIn** consists of a **Point** (*hasPoseLocation_Point*), a **Vector** for the Z axis (*hasPoseLocation_ZAxis*), a **Vector** for the X axis (*hasPoseLocation_XAxis*), and a reference object (*hasPhysicalLocation_RefObject*) all inherited from **PoseLocation**. The data for the **Point**, the Z axis and the X axis are expressed relative to the coordinate system of the reference object. A **PoseLocationIn** indicates that a **SolidObject** that has the **PoseLocationIn** as the value of a *hasSolidObject_PrimaryLocation* or *hasSolidObject_SecondaryLocation* property is inside the **SolidObject** that is the reference object of the **PoseLocationIn**. The notion of 'inside' is vague and might be made more precise.

- **Pose Location On** - A **PoseLocationOn** is a **PoseLocation**. A **PoseLocationOn** consists of a **Point** (*hasPoseLocation_Point*), a **Vector** for the Z axis (*hasPoseLocation_ZAxis*), a **Vector** for the X axis (*hasPoseLocation_XAxis*), and a reference object (*hasPhysicalLocation_RefObject*) all inherited from **PoseLocation**.

The data for the **Point**, the Z axis and the X axis are expressed relative to the coordinate system of the reference object. A **PoseLocationOn** indicates that a **SolidObject** that has the **PoseLocationOn** as the value of a *hasSolidObject_PrimaryLocation* or *hasSolidObject_SecondaryLocation* property is on top of the **SolidObject** that is the reference object of the **PoseLocationIn**. The notion of 'on top of' is vague and might be made more precise.

- **Physical Location** - A **PhysicalLocation** is a **DataThing**. A **PhysicalLocation** says where a **SolidObject** is. A **PhysicalLocation** has a reference object (*hasPhysicalLocation_RefObject*). A **PhysicalLocation** is either a **RelativeLocation** or a **PoseLocation**.

- **Point** - A **Point** is a **DataThing**. A **Point** has X (*hasPoint_X*), Y (*hasPoint_Y*), and Z (*hasPoint_Z*) Cartesian coordinates that are of type **decimal**.

- **Relative Location** - A **RelativeLocation** is a **PhysicalLocation**. A **RelativeLocation** indicates that one **SolidObject** is on or in another **SolidObject**. A **RelativeLocation** must be a **RelativeLocationIn** or a **RelativeLocationOn**. A **RelativeLocation** has a description (*hasRelativeLocation_Description*) that is a **string**.

- **Relative Location In** - A **RelativeLocationIn** is a **RelativeLocation**. A **RelativeLocationIn** indicates that a **SolidObject** that has the **RelativeLocationIn** as the value of a *hasSolidObject_PrimaryLocation* or *hasSolidObject_SecondaryLocation* property is inside the **SolidObject** that is the reference object of the **RelativeLocationIn**. The notion of 'inside' is vague and might be made more precise. A **RelativeLocationIn** has a description inherited from **RelativeLocation** (*hasRelativeLocation_Description*) that is a **string**.

- **Relative Location On** - A **RelativeLocationOn** is a **RelativeLocation**. A **RelativeLocationOn** indicates that a **SolidObject** that has the **RelativeLocationOn** as the value of a *hasSolidObject_PrimaryLocation* or *hasSolidObject_SecondaryLocation* property is on top of the **SolidObject** that is the reference object of the **RelativeLocationOn**. The notion of 'on top of' is vague and might be made more precise. A **RelativeLocationOn** has a description inherited from **RelativeLocation** (*hasRelativeLocation_Description*) that is a **string**.

- **Robot** - A **Robot** is a **SolidObject**. A **Robot** currently has a description of type **string** (*hasRobot_Description*), a work volume that is the union of one to many volumes of type **BoxVolume** (*hasRobot_WorkVolume*), zero or one **EndEffector** (*hasRobot_EndEffector*), and a maximum load weight of type **positiveDecimal** (*hasRobot_MaximumLoadWeight*). A **Robot** belongs to a **KittingWorkstation** (*hadByRobot_Workstation*). The **Robot** ontology provided 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.

- **Shape Design** - A **ShapeDesign** is a **DataThing**. A **ShapeDesign** has only a **Description** (*hasShapeDesign_Description*) that is an **string**. Currently, there is only one derived type of **ShapeDesign**. That is **BoxyShape**.

- **Solid Object** - A **SolidObject** is a **Thing**. A **SolidObject** has a **Location** (*hasSolidObject_PrimaryLocation*). This is a **PhysicalLocation** that relates the location of the object to the location of some other **SolidObject**. The location of a **SolidObject** may be on a **WorkTable** (*hadBySolidObject_WorkTable*). No **SolidObject** except the **Workstation** may be located with respect to itself, and all chains of primary location must end at the **Workstation**. A **SolidObject** may have zero to many **SecondaryLocation** descriptions (*hasSolidObject_SecondaryLocation*). These are also **PhysicalLocations**. The **SecondaryLocations** are required to be logically and mathematically consistent with the value of *hasSolidObject_PrimaryLocation* so that all locations of a **SolidObject** describe (or are consistent with) a single place in space.

- **Stock Keeping Unit** - A **StockKeepingUnit** is a **DataThing**. A **StockKeepingUnit** is a description of a type of object. Every **StockKeepingUnit** has a description of type **string** (*hasSku_Description*), a shape (*hasSku_Shape*), that is a **ShapeDesign**, a weight of type **positiveDecimal** (*hasSku_Weight*), and references to the ids of **EndEffectors** that can handle it (*hasSku_EndEffectors*). A **StockKeepingUnit** belongs to a **KittingWorkstation** (*hadBySku_Workstation*).

- **Vacuum Effector** - A **VacuumEffector** is an **EndEffector**. A **VacuumEffector** holds an object by putting a cup against the object and applying a vacuum. A **VacuumEffector** is either a **VacuumEffectorSingleCup** or a **VacuumEffectorMultiCup**. A **VacuumEffector** has a cup diameter (*hasVacuumEffector_CupDiameter*) and a length (*hasVacuumEffector_Length*) both of which are **positiveDecimals**.

- **Vacuum Effector MultiCup** - A **VacuumEffectorMultiCup** is a **VacuumEffector** with two or more identical cups (*hasMultiCup_ArrayNumber*). A **VacuumEffectorMultiCup** has an array radius of type **positiveDecimal** (*hasMultiCup_ArrayRadius*). 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.

- **Vacuum Effector Single Cup** - A **VacuumEffectorSingleCup** is a **VacuumEffector** with 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** - A **Vector** is a **DataThing**. It has I (*hasVector_I*), J (*hasVector_J*), and K (*hasVector_K*) components that are of type **decimal**.

- **Work Table** - A **WorkTable** is a **BoxyObject**. The top of a **WorkTable** is a flat, rectangular, horizontal surface. The length and width of the top are those of the **BoxyObject**. A **WorkTable** has solid objects that are located with respect to the **WorkTable**, i.e. the reference object of each of those solid objects is the **WorkTable** (*hasWorkTable_SolidObjects*). Typically, those objects will be on top of the **WorkTable**. This property may be deduced by finding all the objects located with respect to the **WorkTable**, so care will be required to keep the values of the *hasWorkTable_SolidObjects* and *hasPhysicalLocation_RefObject* properties consistent. A **WorkTable** belongs to a **Workstation** (*hadByWorkTable_Workstation*).

4. FOR MORE INFORMATION

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

Dr. Stephen Balakirsky

stephen.balakirsky@nist.gov

301-975-4791

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.