

# Outline

- 1) SB - Description of kitting
- 2) Test methods
  - a) TK - Input format (xml, owl)
  - b) TK - Output format (Robot canonical command language)
  - c) SB - method itself (build kits) with varying part mix and configuration, missing parts, ...
- 3) metrics
  - a) TK - Plan
    - i) number of steps
    - ii) goal achievement
    - iii) optimality
    - iv) maintain constraints
    - v) number of errors
  - b) ZK - Execution
- 4) assumptions
  - a) Work volume totally reachable

# Metrics and Test Methods for Industrial Kit Building

Stephen Balakirsky, Thomas Kramer, and Zeid Kootbally

**Abstract**—The IEEE RAS Ontologies for Robotics and Automation Working Group is dedicated to developing a methodology for knowledge representation and reasoning in robotics and automation. As part of this working group, the Industrial Robots sub-group is tasked with studying industrial applications of the ontology. One of the first areas of interest for this subgroup is the area of kit building or kitting. It is anticipated that utilization of the ontology will allow for the development of higher performing kitting systems. However, the definition of "higher performing" has yet to be defined. This paper addresses this issue by providing the basis for performance methods and metrics that are designed to determine the performance of a kitting system.

## I. INTRODUCTION

Material feeding systems are an integral part of today's assembly line operations. These systems assure that parts are available where and when they are needed during the assembly operations by providing either a continuous supply of parts at the station, or a set of parts (known as a 'kit') that contains the required parts for one or more assembly operations. In continuous supply, a quantity of each part that may be necessary for the assembly operation are stored at the assembly station. If multiple versions of a product are being assembled, (mixed-model assembly) a larger variety of parts than are used for an individual assembly may need to be stored. With this material feeding scheme, parts storage and delivery systems must be duplicated at each assembly station.

An alternative approach to continuous supply is known as kitting. In kitting, parts are delivered to the assembly station in kits that contain the exact parts necessary for the completion of one assembly object. According to Bozer and McGinnis [1] "A kit is a specific collection of components and/or subassemblies that together (i.e., in the same container) support one or more assembly operations for a given product or shop order". In the case of mixed-model assembly, the contents of a kit may vary from product to product. Kitting itself may be viewed as a specialization of the general bin-picking problem. The use of kitting allows a single delivery system to feed multiple assembly stations.

In industrial assembly of manufactured products, kitting is often performed prior to final assembly. Manufacturers utilize kitting due to its ability to provide cost savings

[2] including saving manufacturing or assembly space [10], reducing assembly workers walking and searching times [11], and increasing line flexibility [1] and balance [8].

Several different techniques are used to create kits. A kitting operation where a kit box is stationary until filled at a single kitting workstation is referred to as *batch kitting*. In *zone kitting*, the kit moves while being filled and will pass through one or more zones before it is completed. This paper focuses on batch kitting processes.

In batch kitting, the kit's component parts may be staged in containers positioned in the workstation or may arrive on a conveyor. Component parts may be fixtured, for example placed in compartments on trays, or may be in random orientations, for example placed in a large bin. In addition to the kit's component parts, the workstation usually contains a storage area for empty kit boxes as well as completed kits.

Kitting has not yet been automated in many industries where automation may be feasible. Consequently, the cost of building kits is higher than it could be. We are addressing this problem by proposing performance methods and metrics that will allow for the unbiased comparison of various approaches to building kits in an agile manufacturing environment. The performance methods that we propose must be simple enough to be repeatable at a variety of testing locations, but must also capture the complexity inherent to variants of kit building. The test methods must address concerns such as measuring performance against variations in kit contents, kit layout, and component supply. For our test methods, we assume that a robot performs a series of pick-and-place operations in order to construct the kit. These operations include:

- 1) Pick empty kit and place on work table.
- 2) Pick multiple component parts and place in kit.
- 3) Pick completed kit and place in full kit storage area.

Each of these actions may be a compound action that includes other actions such as end-of-arm tool changes, path planning, and obstacle avoidance.

It should be noted that multiple kits may be built simultaneously. Finished kits are moved to the assembly floor where components are picked from the kit for use in the assembly procedure. The kits are normally designed to facilitate component picking in the correct sequence for assembly. Component orientation may be constrained by the kit design in order to ease the pick-to-assembly process. Empty kits are returned to the kit building area for reuse.

S. Balakirsky is with the Intelligent Systems Division, National Institute of Standards and Technology, Gaithersburg, MD, USA (e-mail: stephen.balakirsky@nist.gov)

Z. Kootbally is with the Department of Mechanical Engineering, University of Maryland, College Park, MD, USA (email: zeid.kootbally@nist.gov)

T. Kramer is with the Department of Mechanical Engineering, Catholic University of America, Washington, DC, USA (email: thomas.kramer@nist.gov)

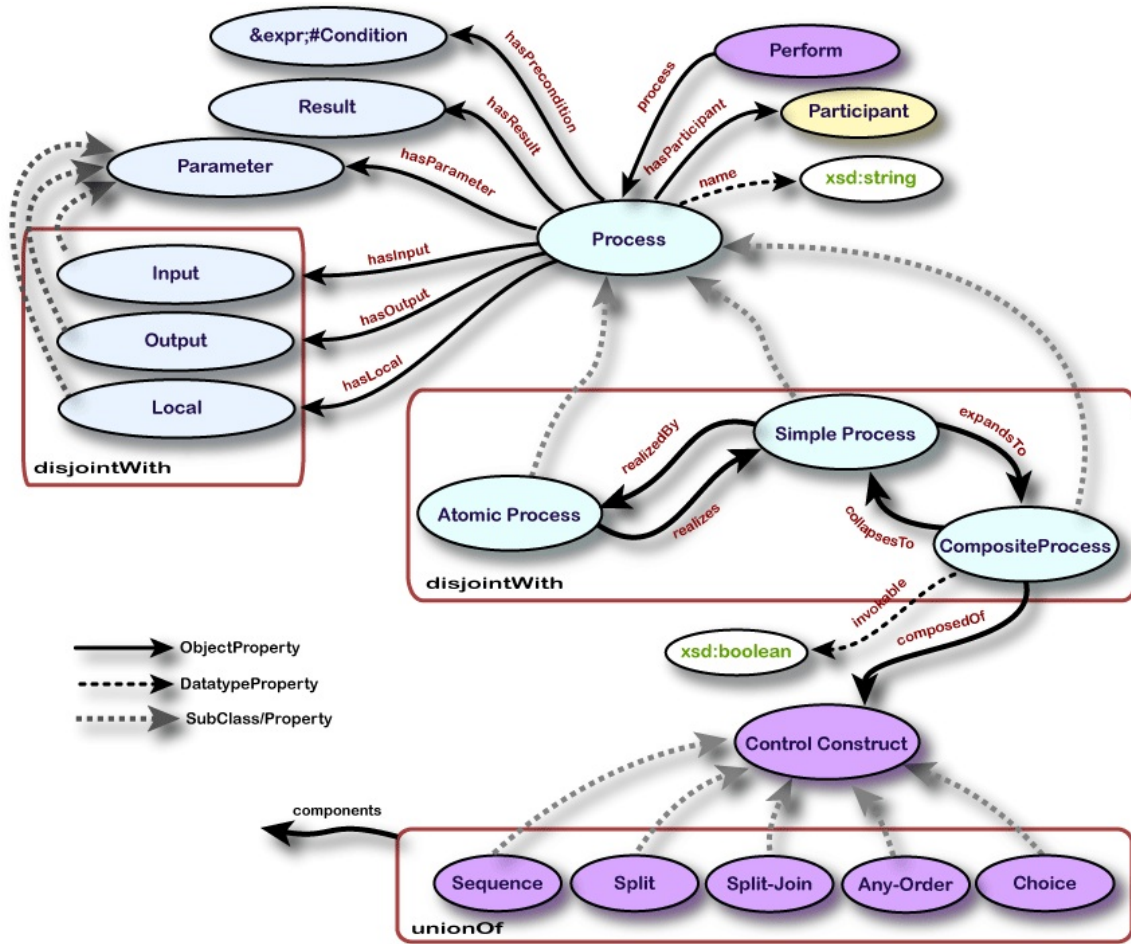


Fig. 1. Kitting data flow abstraction.

## II. TEST METHODS

### A. Overview

Since planning for different kits is a major problem area in building a flexible kitting workstation, the authors are focusing on planning for kitting. A test method is being developed that will be suitable for comparing the performance of different kitting planning systems. To build such a test method, representations for three sets of data are needed:

- a representation for the initial conditions in the kitting workstation from which planning starts (the initial state)
- a representation for the desired final conditions in the kitting workstation after the plan has been executed (the goal state)
- a representation for a plan to get from the initial state to the goal state.

The first two representations are of the same nature: a description primarily of objects and their locations. Hence, the same representation may be used for both. Details follow shortly.

The representation of a plan is of a different nature. A plan is primarily a description of actions that change one kitting workstation state to another. Since the only active element in the model of a kitting workstation is a one-armed robot,

the plan model is a sequential list of actions for a robot to perform.

### B. Kitting Workstation Model

Conceptually, the kitting workstation model is an object model as found in several object oriented programming languages (C++, for example [12]). That is:

- the model consists primarily of class definitions
- a class defines a type of thing
- classes have attributes (“elements” in XML schema language)
- the class definition gives the class (or data type) of each attribute
- some attributes may occur optionally or multiple times
- some classes are derived from others; thus, there is a derivation hierarchy
- a derived class has all the attributes of its parent plus, possibly, some of its own
- if class B is derived from class A, then if the type of an attribute is class A, an instance of class B may be used as the value of the attribute.
- the model does not use multiple inheritance
- the model also uses primitive data types such as numbers and strings and provides for defining specialized

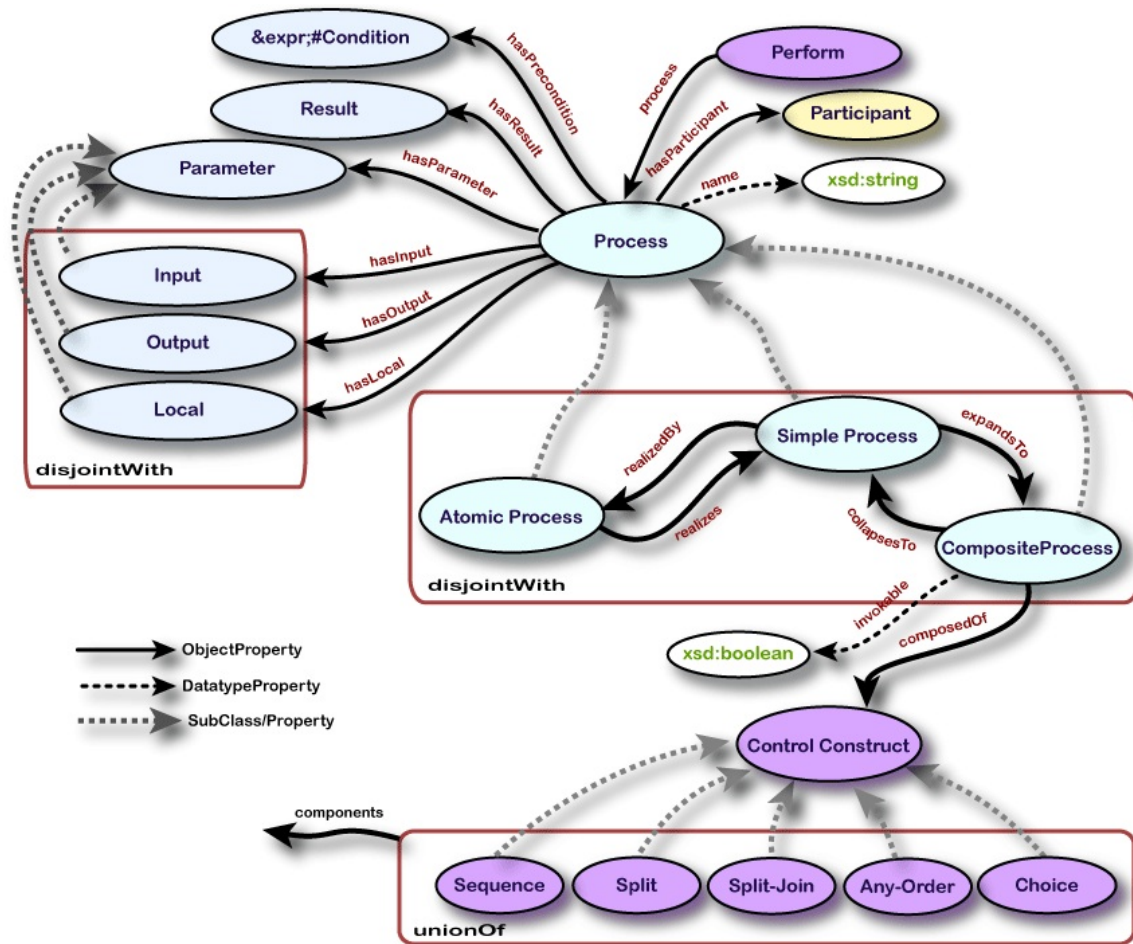


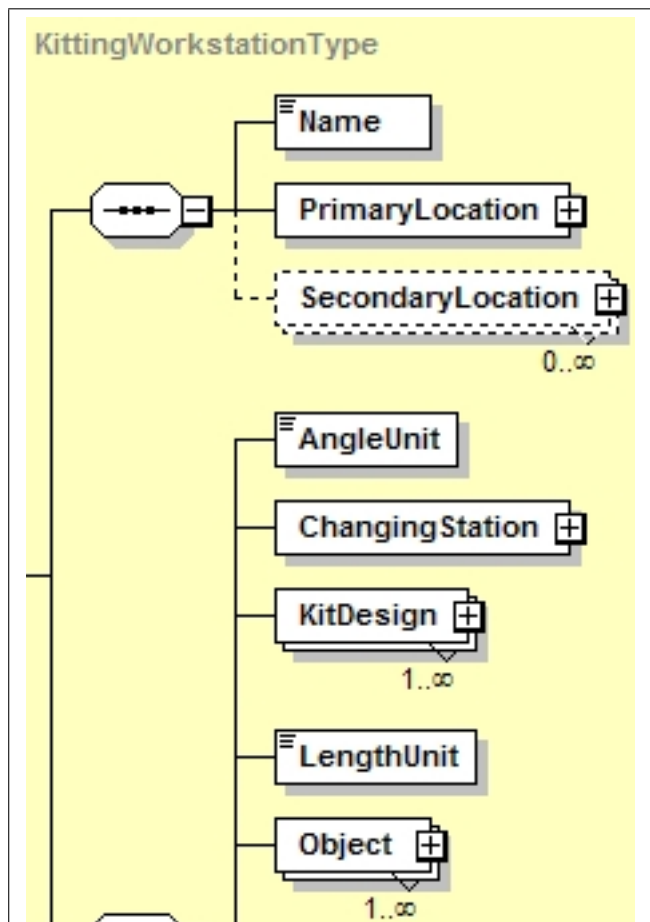
Fig. 2. wow.

data types by putting constraints on primitive data types

A complete hierarchical list of the classes used in the kitting workstation model is shown in Figure 3. In the list, there are two top-level classes, SolidObject and DataThing. All other classes are derived. Each class that is indented in the list is derived from the first less indented class above it. For example, PartsBin is derived from BoxyObject, and BoxyObject is derived from SolidObject. The figure does not show any attributes.

The structure of the kitting workstation class (or type) is shown in Figure 4. The figure shows the names of the attributes of a kitting workstation. The first three attributes (Name, PrimaryLocation and SecondaryLocation) are inherited from the SolidObject class. The rest of the attributes are specific to the kitting workstation 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 Figure 4 and similar figures (which were generated by XMLSpy from XML schemas), a dotted line around a box means the attribute is optional (may occur zero times), while a ..infinity underneath a box means it may occur more than once, with no upper limit on the number of occurrences.



```

SolidObject
  BoxyObject
    KitTray
    LargeContainer
    PartsBin
    PartsTray
    WorkTable
  EndEffector
    GripperEffector
    VacuumEffector
      VacuumEffectorMultiCup
      VacuumEffectorSingleCup
  EndEffectorHolder
  Kit
  KittingWorkstation
  LargeBoxWithEmptyKitTrays
  LargeBoxWithKits
  Part
  PartsTrayWithParts
  Robot
DataThing
  BoxVolume
  KitDesign
  PartRefAndPose
  PhysicalLocation
    PoseLocation
      PoseLocationIn
      PoseLocationOn
      PoseOnlyLocation
  RelativeLocation
    RelativeLocationIn
    RelativeLocationOn
  Point
  ShapeDesign
  StockKeepingUnit
  Vector

```

Fig. 3. Kitting Workstation Model Class Hierarchy

Fig. 4. Kitting Workstation Model

The types (i.e. classes or datatypes) of the attributes of a kitting workstation are not shown on Figure 4. The structures of several of the attributes are shown on the following figures:

- ChangingStation – EndEffectorChangingStationType: Figure 5
- KitDesign – KitDesignType: Figure 6
- Object – LargeBoxWithEmptyKitTraysType Figure 7, LargeBoxWithKitsType Figure 8, and PartsTrayWithParts Figure 9
- Robot – RobotType Figure 10
- Sku – StockKeepingUnitType Figure 11
- WorkTable – WorkTableType Figure 12.

The type of the Object elements of a kitting workstation is SolidObject. That is an abstract class not intended to be instantiated. Hence, the figures show the structures of derived classes of SolidObject that are intended to be used for instances of the Object attribute.

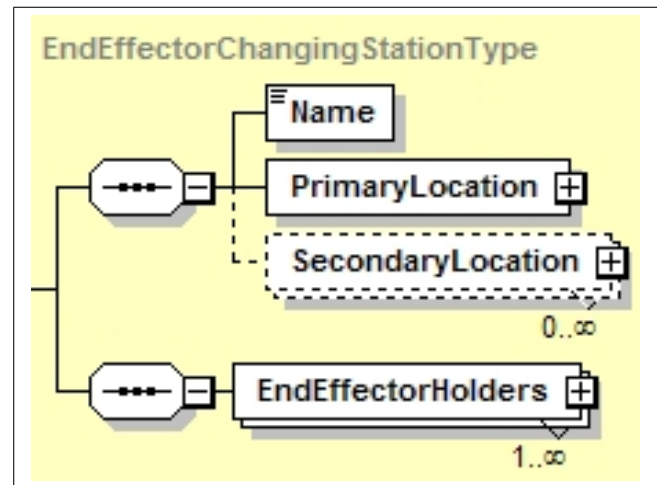


Fig. 5. Changing Station Model

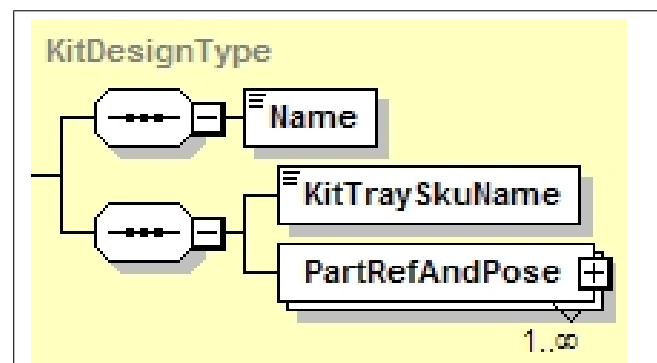


Fig. 6. Kit Design Model

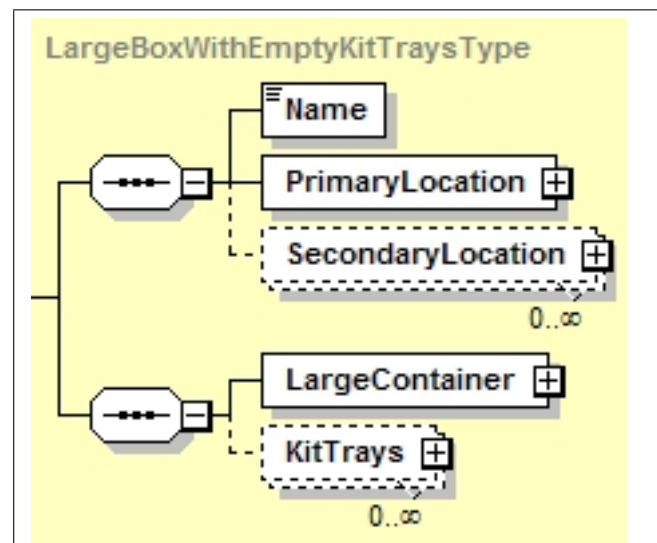


Fig. 7. Large Box With Empty Kit Trays Model

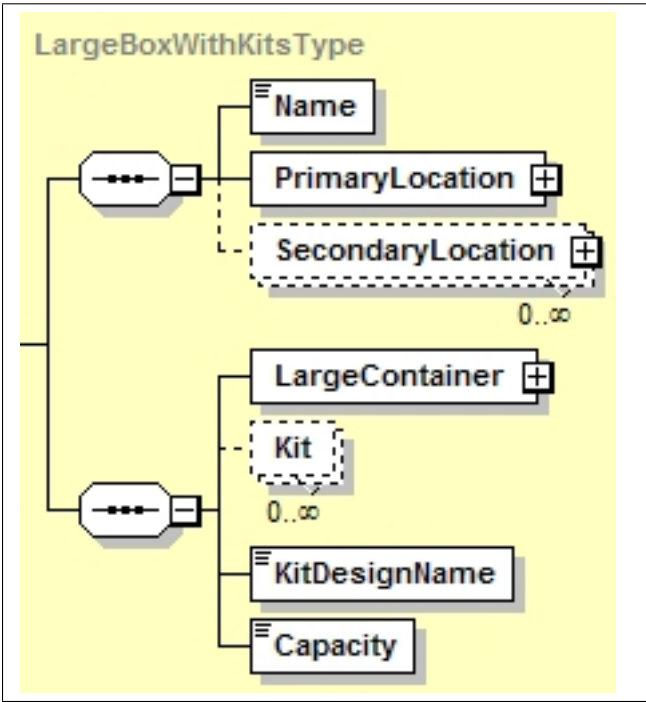


Fig. 8. Large Box With Kits Model

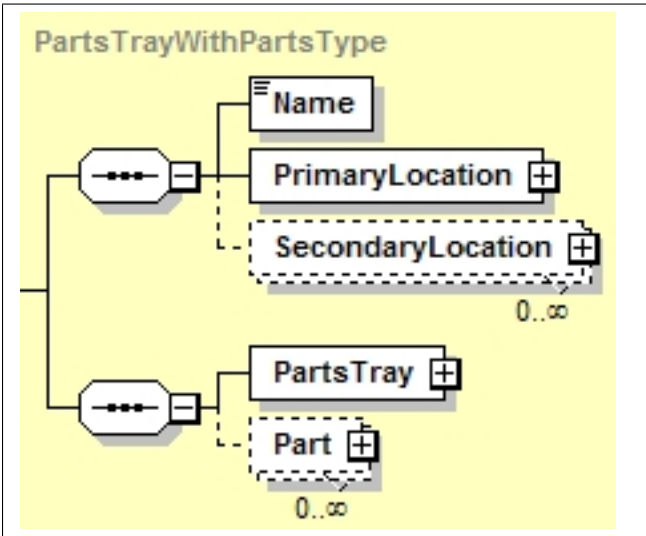


Fig. 9. Parts Tray With Parts Model

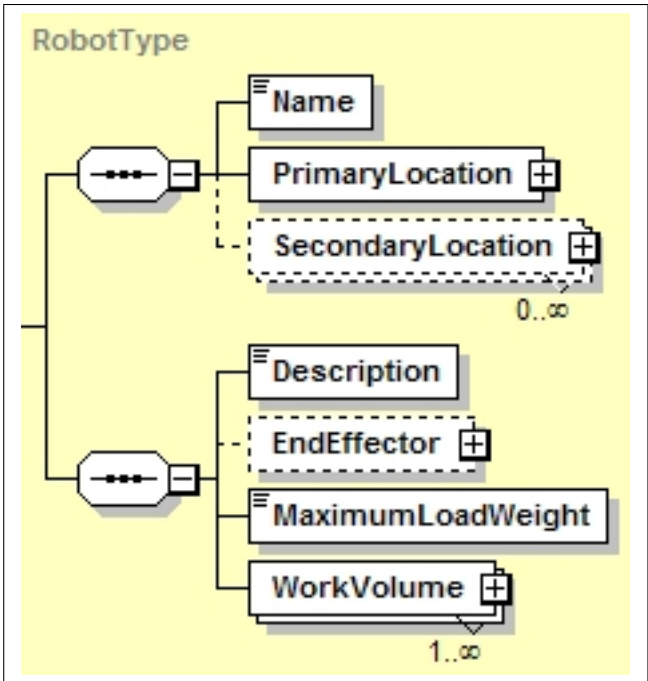


Fig. 10. Robot Model

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.

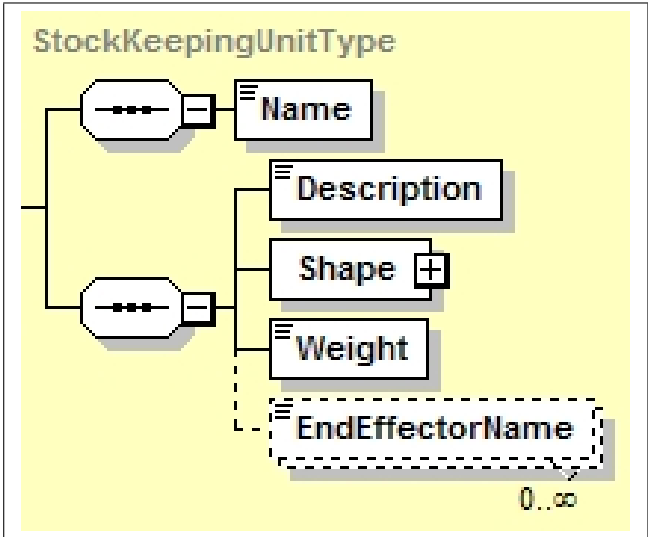


Fig. 11. Stock Keeping Unit Model



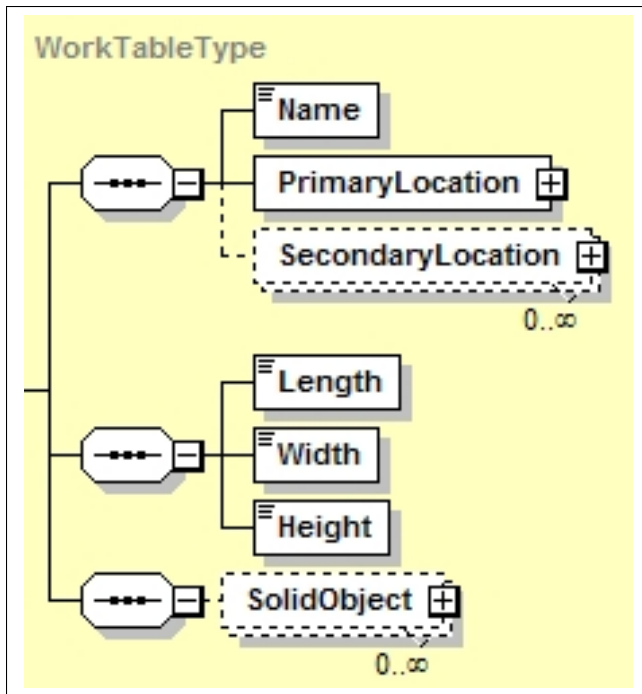


Fig. 12. Work Table Model

The kitting workstation model has been fully defined in each of two languages: XML schema language [18], [13], [14], and Web Ontology Language (OWL *sic*) [17], [15], [16]. 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 movable objects that are in the kitting workstation are explicitly modeled in the XML schema model but not in the OWL model.
- Attribute names in OWL have a prefix, as described below. The prefixes are not used in XML schema.

### C. Using OWL

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.

In addition to having the model defined in OWL, OWL data files describing specific initial states and goal states

were defined in OWL, also using the functional-style syntax. Software tools were built in C++ to work with the OWL model and data files conforming to the model.

The initial intent has been to use OWL files for presenting the initial and goal conditions for planning problems, and the authors have implemented a planning system that uses OWL files.

The primary tool used by the OWL community for building and checking OWL models and data files is named Protege [6]. Protege was used for checking the kitting model and data files as they were built. Protege continues to be used for checking the model and data files whenever they are changed. The layout of the hierarchy in Figure 3 is identical to what may be seen in Protege’s class hierarchy window when the kitting model is loaded (but Protege makes it look better).

Several difficulties of working with OWL were encountered. The primary effect of these difficulties was to make it essentially impossible to debug OWL files.

Defining a model in OWL is quite different from defining the same model in other information modeling languages with which the authors are intimately familiar: C++, EXPRESS [7], and XML schema. Three of the major differences involve (1) the assignment of attributes in classes, (2) OWL’s “open world” assumption, and (3) the distinction between model files and data files.

1) *Class Attributes:* In other languages, assigning a typed attribute to a class requires a single line of code. For example, the x attribute may be put into a cartesian point class in XML schema language with

```
<xs:element name="X" type="xs:decimal"/>
```

or in C++ with

```
double x;
```

or in EXPRESS with

```
x : REAL;
```

In these other languages, the name of the attribute is local to the class. Hence, an attribute with a given name can appear in more than one class, and there will be no confusion.

In OWL, there is no simple method of declaring a class attribute. Instead, a property must be declared along with properties of the property. The following lines are used in the OWL model to say that all points and only points have an x attribute which is a decimal number.

```
Declaration(DataProperty(hasPoint_X))
DataPropertyDomain(:hasPoint_X :Point)
DataPropertyRange(:hasPoint_X xsd:decimal)
EquivalentClasses(:Point ObjectIntersectionOf(
DataSomeValuesFrom(:hasPoint_X xsd:decimal)
DataAllValuesFrom(:hasPoint_X xsd:decimal)))
```

The hasPoint prefix used in the property name is not an OWL requirement. It is one of several naming conventions for OWL being used by the authors. The prefix is both for the benefit of a human reader (to make it obvious that this is a property of a Point) and to differentiate this x attribute

from an `x` attribute of some other class (call it `Foo`) which would have the prefix `hasFoo`.

As described above, with OWL, it is necessary to make many statements in order to build a class in a typical object-oriented style. OWL does not assume a typical object-oriented style. It assumes the world might be more complex than that. Hence, many OWL statements are required to produce effects made in a few statements in other object-oriented languages. Having to write a lot of statements is tedious but not a roadblock. A more serious problem is that if a statement necessary to produce an object-oriented effect is omitted, that is not an OWL error. Protege does not have an object-oriented mode in which it will warn the user if a required statement is missing. There are no OWL tools that will help with finding missing statements. This is a debugging problem.

OWL was built so that it would support automated reasoning about the relationships among properties, classes, and individuals. Protege allows the use of several alternate automatic reasoners. In a typical object-oriented style, there is no use for reasoning of that sort. Everything useful to know about the relationships among properties, classes, and individuals is already known. Hence having an automated reasoning capability of the sort for which OWL was built is not useful for the kitting model.

2) *Open World Assumption:* OWL makes an “open world” assumption. In an open world, anything might be true that is not explicitly declared false and is not inconsistent with what has been declared true. This makes it easy for errors to go unrecognized as such by Protege (or any other OWL tool). For example, suppose the line `DataPropertyDomain(:hasPoint_X :Point)` given above is mistyped as `DataPropertyDomain(:hasPoint_x :Point)`. When Protege loads the file and the reasoner is started, no errors are detected. Protege assumes that the `DataPropertyDomain` for `hasPoint_X` is unknown (that’s fine in OWL and Protege) and that there is a new property named `hasPoint_x` about which the only thing known is its `DataPropertyDomain` (also fine in OWL and Protege, even though there is no explicit `DataProperty` declaration for the new property). The error can be detected by a human by studying the list provided by selecting the `DataProperties` tab in Protege, but the human should not have to go to such lengths. Similar errors, such as mistyping the name of an individual, are similarly accepted in OWL and Protege as being OK, with similar effects. The difficulties caused by the open world assumption would not occur if Protege had a closed world mode, but it has none.

3) *Model Files vs. Data Files:* While other languages have different file formats for models and data conforming to the models, OWL does not distinguish between model files and data files. Protege does not provide any method of specifying that a file is a model file or a data file. The conceptual difference is simple, even in OWL. Model files describe classes and data types (and, possibly, constraints).

Data files give information about individuals (instances of one or more classes – often called objects). The authors have made it a practice to distinguish OWL model files from OWL data files. An OWL data file can inadvertently change an OWL model, a bug that is very hard to find. That cannot happen with EXPRESS or XML schema.

4) *Bugs in Files:* Since humans are error-prone, and the kitting OWL files were built by humans, the OWL files had errors of the sort mentioned above. Some of these errors were discovered when the OWL files were processed by the tools developed for processing them and strange results were observed. Other errors were found when a method of generating OWL data files automatically from XML data files was developed, as described next.

#### *D. Using XML Schema and XML Data Files*

To work around the difficulties of using OWL while continuing to use OWL, the authors are using XML schema and XML data files in parallel with the corresponding OWL files.

1) *XML Tools:* Two automated tools developed by the authors are being used: an xml schema parser (`xmlSchemaParser`) and a code generator (`GenXMiller`). Neither of these has yet been documented elsewhere.

The `xmlSchemaParser` reads an XML schema file, stores it in terms of instances of C++ classes, and reprints the schema. When the `xmlSchemaParser` runs, it performs many checks on the validity of the schema that is input to it. The `xmlSchemaParser` handles almost all portions of the XML schema syntax. A few of the rarely-used bits of syntax are not implemented.

The `GenXMiller` reads an XML schema and writes code for reading and writing XML data files corresponding to that schema. The code that is generated includes C++ classes (`.hh` and `.cc` files), a parser (YACC and Lex files) and a stand-alone parser file in C++ that uses the other files. The executable utility produced by compiling a stand-alone parser reads and echos any XML data file corresponding to the schema. The `GenXMiller` is still under development. It handles many schemas, but has not implemented enough of the XML schema language to handle many others. The `GenXMiller` is not a new type of system. Several other code generators that use an XML schema as input have been developed. Even more XML schema parsers are available. However, having the knowledge about XML schema and XML data files gained by developing that software and having an intimate knowledge of the source code for it has proved very valuable in using XML to deal with OWL.

The `xmlSchemaParser` and the `GenXMiller` use the same underlying parser, which is built in YACC and Lex [9].

In addition to using the `xmlSchemaParser` and the `GenXMiller`, a commercial XML tool named `XMLSpy` [5] has been used to check all XML schemas and XML data files.



2) *Handling Kitting Data Files*: There is only one conceptual kitting model but there are several kitting data files corresponding to it, and, if the kitting model is used, there will be many more data files. Hence, the problem of generating bug-free data files was tackled first.

An XML schema, *kitting.xsd*, was written by hand modeling the same information as the OWL kitting workstation model, *kittingClasses.owl*. The GenXMiller was then used to generate C++ classes and a parser for XML kitting data files corresponding to *kitting.xsd*. The C++ classes that were generated, of course, included code for printing XML kitting data files. That code was rewritten by hand so that it prints OWL data files rather than XML data files. The utility produced by compiling the code is called the *owlPrinter*. To produce an OWL kitting data file, one writes an XML kitting data file and runs it through the *owlPrinter*.

To determine that the *owlPrinter* works properly, it seems sufficient to demonstrate that OWL data files generated automatically by the *owlPrinter* from XML data files conforming to *kitting.xsd* contain exactly the same OWL statements as are contained in manually prepared OWL data files intended to contain the same information and conforming to *kittingClasses.owl*. This demonstration was achieved as follows.

- 1) Three XML data files were written manually containing the same information as three OWL data files. Each of the OWL files was at least 1100 lines (20 pages) long. Among the three there were statements of almost all sorts possible under the *kittingClasses.owl* model. It was decided, therefore, that successful performance for these three files would be an adequate test.
- 2) The three XML data files were run through the *owlPrinter* to produce three OWL files.
- 3) Since the *owlPrinter* has a different approach to ordering OWL statements as was taken in preparing OWL files manually and a slightly different method of formatting statements, two small utilities were written to enable file comparison. The first utility, *compactOwl*, reads an OWL file and writes an OWL file containing the same statements but with blank lines and comments removed, and with each statement on a single line. For each pair of matching OWL files (manually written and automatically generated), *compactOwl* was used to generate a corresponding pair of compacted OWL files. The second utility, *compareOwl*, reads each of a pair of OWL files, alphabetizes the statements from each of them on two saved lists, and then goes through the two lists checking that the *n*th line of one list is identical to the *n*th line of the other list. *CompareOwl* was used to compare each of the three sets of pairs of compacted files.
- 4) While the tests just described were being made, changes were made to correct errors in the manually written XML and OWL data files being tested and in the code for the *owlPrinter*. The tests revealed errors in all three types of files. In particular, errors in the manually written OWL data files were found that had not been detected any other way.

After the testing just described was complete, using the *owlPrinter*, another OWL data file was prepared from a manually written XML data file for which there was no manually written OWL counterpart. The automatically generated OWL data file was checked in Protege and no errors were reported – although as described earlier, being OK in Protege does provide much assurance of correctness.

Once the *owlPrinter* has been debugged, as it appears to be, OWL data files may be prepared with much less likelihood of error for the following reasons.

- Property names and names of individuals will not be misspelled.
- Statements will not be accidentally omitted.
- Validity checks made in the *kittingParser* and *XML-Spy* will do a better job of detecting errors in XML data files. For example, required attributes that are missing will be detected.

3) *Handling the Kitting Model*: As described above, the equivalent model files *kitting.xsd* file and *kittingClasses.owl* were both prepared manually. If changes to the kitting model are made, it will be necessary to change both of those files and the code for the *owlPrinter*. It would be good to have *kitting.xsd* as the primary source file for the model and to generate *kittingClasses.owl* automatically from it. The authors believe this is possible and have started working on it. The work is not yet complete, but no roadblocks are in sight. The approach being used is to modify the printer code in the *xmlSchemaParser* so that it prints an OWL class file rather than an XML schema file.

It would also be desirable to be able to modify the *owlPrinter* automatically if the kitting model is changed. Doing that is a substantially more difficult task than the other two automatic conversions, and the authors are not planning to attempt it. The approach would be to modify the GenXMiller so that the code it generates automatically would read XML data files and automatically generate OWL data files.

#### *E. Plan Model*

As mentioned earlier, the plan format being used is a sequential list of actions for a robot to perform. The authors devised a “canonical robot command” language in which such lists can be written. The purpose of the canonical robot command language (CRCL) is to provide generic commands that implement the functionality of typical industrial robots without being specific either to the language of the planning system that makes a plan or to the language used by a robot controller that executes a plan.

It was anticipated that planning systems would plan in some language used by automated planners and that plans made by such systems would be translated into the canonical robot command language. It was anticipated also that plans would be executed by a variety of robot controllers using robot-specific languages for input programs. The authors themselves are using a PDDL planner [4] to generate plans in PDDL output language and are using a ROS controller [3] to control a robot. Those two systems are connected

using files of robot commands in CRCL. After a plan has been generated by the PDDL planner, the plan is translated into a CRCL file. When the plan is being executed, the CRCL commands are translated into ROS commands.

*1) Canonical Robot Command View of a Robot:* A robot suitable for use with CRCL commands has one arm and can position and orient the end of the arm anywhere in some work volume within some tolerance. At each point in the work volume, the range of orientations that can be attained may be limited.

The speed and acceleration of the end of the arm may be controlled.

A robot can attach one end effector at a time to the end of the arm from an end effector changing station and can detach the end effector at the changing station. The changing station itself is passive. Attaching an end effector is done by (1) moving the robot arm (with no end effector attached) to an attachment position with respect to an end effector and (2) giving a CloseToolChanger command. Detaching an end effector is done by (1) moving the robot arm (with an end effector attached) to a detachment position and (2) giving an OpenToolChanger command. The attachment and detachment positions are normally at an end effector changer in the end effector changing station.

All end effectors available to the robot are stored in the end effector changing station.

All end effectors are assumed to be grippers.

All grippers have two states, open and closed. A gripper can hold an object in the closed state and cannot hold an object in the open state. [Additional states may be added later, such as open a certain distance or closed with a certain force.]

Opening or closing any gripper mounted at the end of a robot arm is exercised by giving a command to the robot.

The robot cannot simultaneously move and open or close the gripper.

There is always a controlled point. When no end effector is on the arm, the controlled point is at the end of the arm. When an end effector is mounted on the end of the arm, the controlled point is the tool centre point.

The robot can move the controlled point smoothly through a series of poses from a start pose at which it is not moving to an end pose at which it is not moving, provided that all poses are given before motion starts. The acceleration and steady state speed of the controlled point may be specified. The robot will do its best to maintain the requested steady state speed but may reduce (but not increase) speed or acceleration as necessary to allow for the dynamics of arm motion.

A tolerance for the intermediate points of a smooth motion may be set. The controlled point must pass the intermediate points within the given tolerance (without coming back to a point after missing it by more than the tolerance).

*2) General Description of CRCL:* CRCL includes commands for a robot controller. In normal system operation,

CRCL commands will be translated into the robot controller's native language by the robot's plan interpreter as it works its way through a CRCL plan. One CRCL command may be interpreted into several native language commands.

The robot controller may put canonical robot commands on a queue and execute them (in order) when it wishes.

If the robot controller is unable to execute a particular instance of a canonical robot command, subsequent behavior is up to the robot controller. The safest course of action to abort execution of the plan.

The pose at the end of a command is called the current pose.

While a plan is being executed, the robot should not move except as directed by a canonical robot command.

Status of command execution is not returned by the robot controller to the plan interpreter (or any other command generator).

The syntax of commands is given below using C++ syntax. The command name is given followed by the command arguments (if any) in parentheses, including the types of the arguments.

The coordinate system for poses used in the canonical robot commands is the workstation coordinate system.

Note that the robot cannot be commanded by canonical robot commands in terms of its joint angles (or distances).

Three of the CRCL commands use the Pose structure. The Pose structure gives the location and orientation of the coordinate system of the controlled object. The controlled object is the gripper if the robot has one attached or the outermost component of the robot arm if not. The location is specified by the point in kitting workstation coordinates at which the origin of the coordinate system of the controlled object lies. The point is described by giving its X, Y, and Z values. The orientation of the controlled object is specified by giving the I, J, and K components in kitting workstation coordinates of the Z and X axes of the coordinate system of the controlled object.

### *3) CRCL Commands*

CloseGripper()

Close the gripper.

CloseToolChanger()

Close the tool changer on the robot so that it attaches to a tool. The robot must be in an appropriate position with respect to the tool for the changer mechanism on the robot to attach to the tool.

Dwell (double time)

Stay motionless for the given amount of time in seconds.

EndCanon(int reason)

Do whatever is necessary to stop executing canonical robot commands. No specific action is required. The robot controller should not execute any canonical robot command except InitCanon after executing EndCanon and should

signal an error if it is given one. This command will normally be given when execution of a plan is complete. It may also be given if the plan interpreter detects an error in the plan or is unable to proceed for any other reason. A value of 0 for reason indicates that execution of a plan has completed successfully. A positive value of reason indicates not.

#### InitCanon()

Do whatever is necessary to get ready to move. Length and angle units are set to the default units. This command will normally be given when the plan interpreter opens a plan to be executed.

#### Message (string message)

Display the given message on the operator console.

#### MoveStraightTo(Pose \* pose)

Move the controlled point in a straight line from the current pose to the given pose, and stop there.

#### MoveThroughTo(Pose \*\* poses, int numPoses)

Move the controlled point along a trajectory passing near all but the last of the given poses, and stop at the last of the given poses. The numPoses gives the number of poses.

#### MoveTo(Pose \* pose)

Move the controlled point along any convenient trajectory from the current pose to the given pose, and stop there.

#### OpenGripper()

Open the gripper.

#### OpenToolChanger()

Open the tool changer on the robot so that it releases the end effector. This is normally done after the end effector attached to the robot has been moved into an end effector changer.

#### SetAbsoluteAcceleration(double acceleration)

Set the acceleration for the controlled point to the given value in length units per second per second.

#### SetAbsoluteSpeed(double speed)

Set the speed for the controlled point to the given value in length units per second.

#### SetAngleUnits(string UnitName)

Set angle units to the unit named by the UnitName. The unit name must be one of “degree” or “radian”. All commands that use angle units (for orientation or orientation tolerance) are in terms of those angle units. Existing values for orientation are converted automatically to the equivalent value in new angle units. The default angle unit is “degree”.

#### SetEndAngleTolerance(double tolerance)

Set the tolerance for the orientation of the end of the arm (whenever there is no gripper there) or of the gripper (whenever a gripper is on the end of the arm) to the given value in current angle units.

#### SetEndPointTolerance(double tolerance)

Set the tolerance for the position of the end of the arm (whenever there is no gripper there) or of the tool centre point (whenever a gripper is on the end of the arm) to the given value in current length units.

#### SetIntermediatePointTolerance(double tolerance)

Set the tolerance for smooth motion near intermediate points to the given value in current length units.

#### SetLengthUnits(string UnitName)

Set length units to the unit named by the UnitName. The unit name must be one of “inch”, “mm” or “meter”. All commands that use length units (for location, tolerance, speed, and acceleration) are in terms of those length units. Existing values for speed, position, acceleration, etc. are converted automatically to the equivalent value in new length units. The default length unit is millimeters, “mm”.

#### SetRelativeAcceleration(double percent)

Set the acceleration for the controlled point to the given percentage of the robot’s maximum acceleration.

#### SetRelativeSpeed(double percent)

Set the speed for the controlled point to the given percentage of the robot’s maximum speed.

#### StartObjectScan(string name)

Activates\* the object sensor, if it isn’t already, and adds the given name to the list of parts being searched for. When the object sensor detects the part, it removes the part from the list of parts to search for and writes a sequence of canonical commands to “output\_commands.txt” (as a placeholder for updating the mySQL database). When the list of parts to search for becomes empty, the object sensor stops motion as soon as possible and the controller ignores further movement commands until the sensor is deactivated.

#### StopObjectScan()

Deactivates\* the object sensor.

\*Activate and deactivate are used loosely, because the controller is still always subscribed to the sensor topic. But, the sensor callback returns immediately if the sensor is deactivated.

#### StopMotion(integer isEmergency)

Stop the robot motion. If isEmergency is not 0, then stop as soon as possible regardless of damage to the system. If isEmergency is 0 then come to a graceful stop.

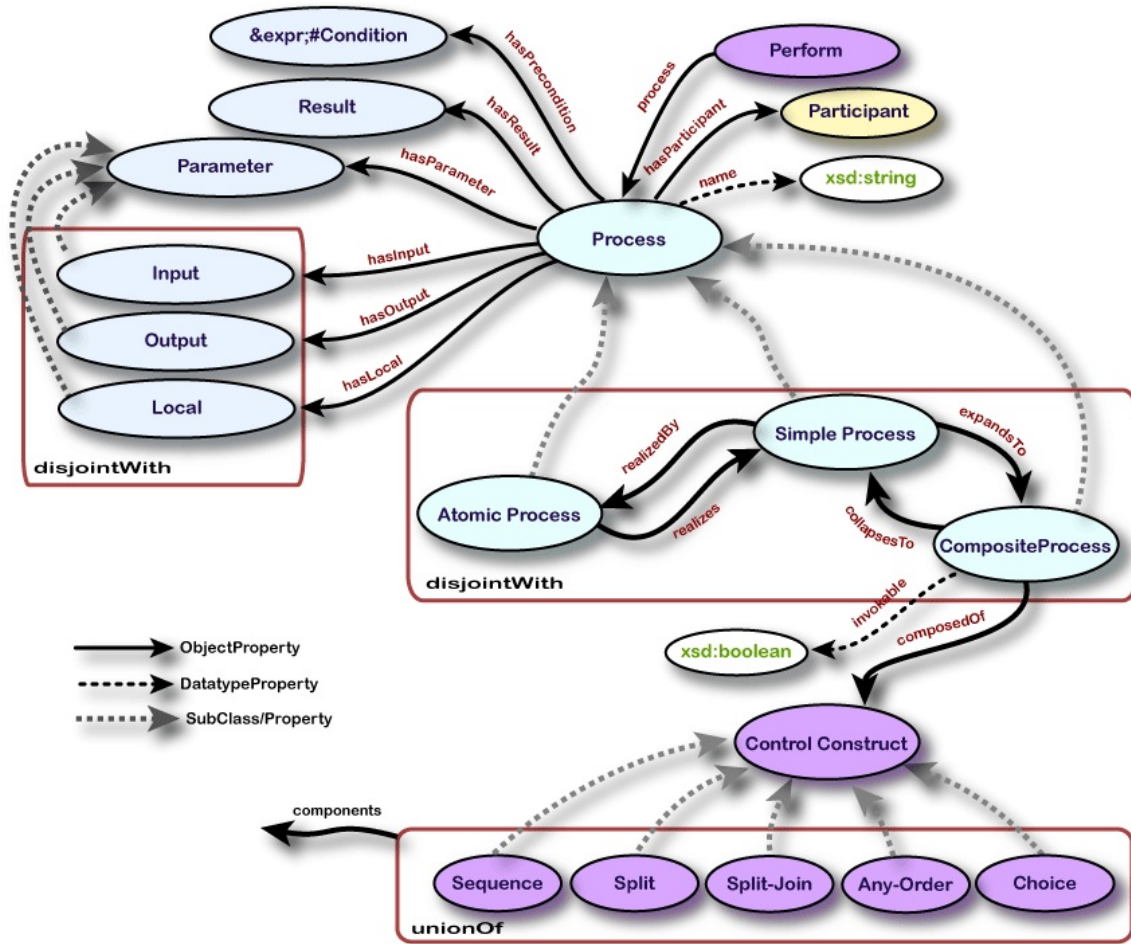


Fig. 13. wow, it works?

### III. METRICS

#### A. Plan Metrics

Blah, Blah, Blah  
more blah blah

#### B. Execution Metrics

Blah, Blah, Blah

### IV. ASSUMPTIONS

Blah, Blah, Blah

### V. CONCLUSIONS AND FUTURE WORK

#### REFERENCES

- [1] Y. A. Bozer and L. F. McGinnis. Kitting versus line stocking: A conceptual framework and descriptive model. *International Journal of Production Economics*, 28:1–19, 1992.
- [2] O. Carlsson and B. Hensvold. Kitting in a High Variation Assembly Line. Master's thesis, Luleå University of Technology, 2008.
- [3] Willow Garage. Robot Operating System (ROS). In <http://www.willowgarage.com/pages/software/ros-platform/>, 2012.
- [4] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—The Planning Domain Definition Language. Technical Report CVC TR98-003/DCS TR-1165, Yale, 1998.
- [5] Altova GMBH. *Altova XMLSpy 2010 User & Reference Manual*. Altova GMBH, Vienna, Austria, 2010.
- [6] M. Horridge. *A Practical Guide To Building OWL Ontologies Using Protege 4 and CO-ODE Tools, Edition 1.3*. University of Manchester, Manchester, England, 2011.
- [7] ISO. *Industrial automation systems and integration – Product data representation and exchange – Part 11: Description method: The EXPRESS language reference manual*. ISO, Geneva, Switzerland, 2003.
- [8] J. Jiao, M. M. Tseng, Q. Ma, and Y. Zou. Generic Bill-of-Materials-and-Operations for High-Variety Production Management. *Concurrent Engineering: Research and Applications*, 8(4):297–321, December 2000.
- [9] J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly, Cambridge, MA, USA, 1995.
- [10] L. Medbo. Assembly Work Execution and Materials Kit Functionality in Parallel Flow Assembly Systems. *International Journal of Production Economics Journal of Industrial Ergonomics*, 31:263–281, 2003.
- [11] G.F. Schwind. How Storage Systems Keep Kits Moving. *Material Handling Engineering*, 47(12):43–45, 1992.
- [12] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, New York, NY, USA, 2000.
- [13] W3C. XML Schema Part 0: Primer Second Edition. In <http://www.w3.org/TR/xmlschema-0/>, 2004.
- [14] W3C. XML Schema Part 1: Structures Second Edition. In <http://www.w3.org/TR/xmlschema-1/>, 2004.
- [15] W3C. OWL 2 Web Ontology Language Primer. In <http://www.w3.org/TR/owl2-primer/>, 2009.
- [16] W3C. OWL 2 Web Ontology Language Structural Specification and Functional Syntax. In <http://www.w3.org/TR/owl2-syntax/>, 2009.
- [17] W3C. OWL 2 Web Ontology Language Document Overview. In <http://www.w3.org/TR/owl-overview/>, 2012.

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