NISTIR 7942

Metrics and Test Methods for Industrial Kit Building

Stephen Balakirsky
Thomas Kramer
Zeid Kootbally
Anthony Pietromartire

http://dx.doi.org/10.6028/NIST.IR.7942

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 7942

Metrics and Test Methods for Industrial Kit Building

Stephen Balakirsky
Thomas Kramer
Zeid Kootbally
Anthony Pietromartire
Intelligent Systems Division
Engineering Laboratory

http://dx.doi.org/10.6028/NIST.IR.7942

May 2013

U.S. Department of Commerce
Rebecca Blank, Acting Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director

Metrics and Test Methods for Industrial Kit Building

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

***Abstract*— The IEEE Robotics and Automation Society's (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 knowledge representation. 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 knowledge representation 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 is 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. The use of kitting allows a single delivery system to feed multiple assembly stations. The individual operations of the station that builds the kits may be viewed as a specialization of the general bin-picking problem [2].

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

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 [3] including saving manufacturing or assembly space [4], reducing assembly workers walking and searching times [5], and increasing line flexibility [1] and balance [6].

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 in 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 up an empty kit and place it on the work table.
2) Pick up multiple component parts and place them in a kit.
3) Pick up the completed kit and place it in the full kit storage area.

Each of these 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.

II. Prerequisites

A. Overview

Planning for different kits is a major problem area in building a flexible kitting workstation. Therefore, one area of focus for the authors is metrics and test methods for 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, certain system prerequisites are necessary for the planning system under test as well as for the hardware that will be utilized in the implementation of the test method. In order to provide for a consistent test metric, the system under test needs a standardized representation for three sets of data:

- 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 are presented in Section II-B.

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 our 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 Data Representation

Conceptually, the kitting workstation model is an object model as found in several object oriented programming languages (C++, for example [7]). 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 for individual variables) 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 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 1. 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, WorkTable is derived from BoxyObject, and BoxyObject is derived from SolidObject. The figure does not show any attributes.

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

Fig. 1. Kitting Workstation Model Class Hierarchy

The structure of the kitting workstation class (or type) is shown in Figure 2. 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 2 and similar figures (which were generated by XMLSpy [1] from XML schemas), a dotted line around a box means the attribute is optional (may occur zero times), while a ..∞ underneath a box means it may occur more than once, with no upper limit on the number of occurrences.

The types (i.e., classes or datatypes) of the attributes of a kitting workstation are not shown in Figure 2. The structures of several of the attributes are shown in the following figures:

- ChangingStation – EndEffectorChangingStationType: Figure 3
- KitDesign – KitDesignType: Figure 4
- Object – LargeBoxWithEmptyKitTraysType: Figure 5, LargeBoxWithKitsType: Figure 6, and PartsTrayWithPartsType: Figure 7
- Robot – RobotType: Figure 8
- Sku – StockKeepingUnitType: Figure 9
- WorkTable – WorkTableType: Figure 10.

The type of the Object elements in a kitting workstation is SolidObject. That is an abstract class not intended to be instantiated. Hence, Figures 3 through 10 show the structures of derived classes of SolidObject that are intended to be used for instances of the Object attribute.

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.

The kitting workstation model has been fully defined in each of two languages: XML schema language [8], [9], [10], and Web Ontology Language (OWL *sic*) [11], [12], [13]. Further information on the implementations may be found in Section III.

C. Robot Requirements

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 (CRCL) in which such lists can be written. The purpose of the canonical robot command language 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 Planning Domain Definition Language (PDDL) planner [14] to generate plans in PDDL output language and are using a ROS controller [15] 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.

In order to support this mode of operation, the basic robot and robotic workcell must meet certain requirements. These include:

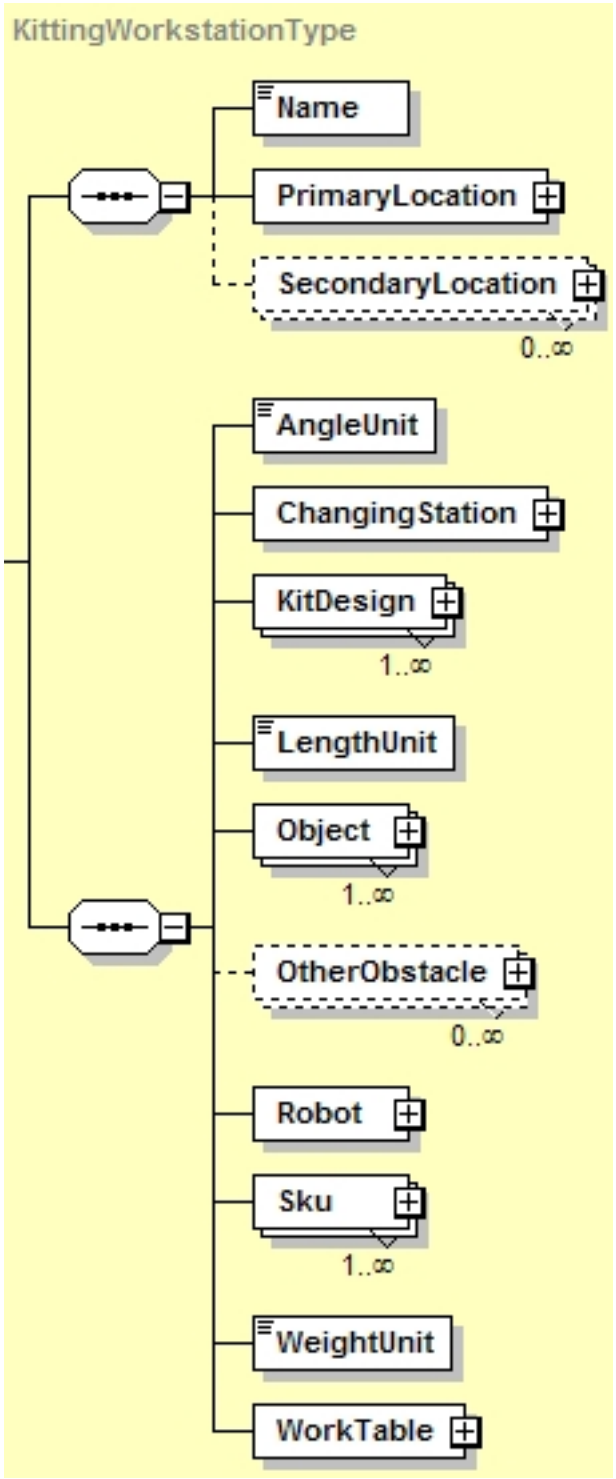

Fig. 2. Kitting Workstation Model

[1]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 or NIST, nor does it imply that the software tools identified are necessarily the best available for the purpose.

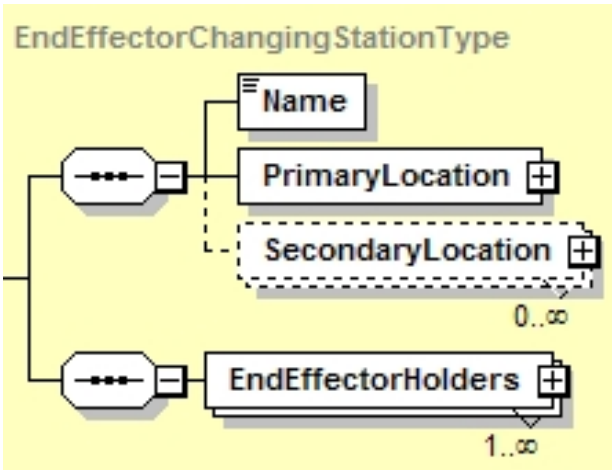

Fig. 3. Changing Station Model

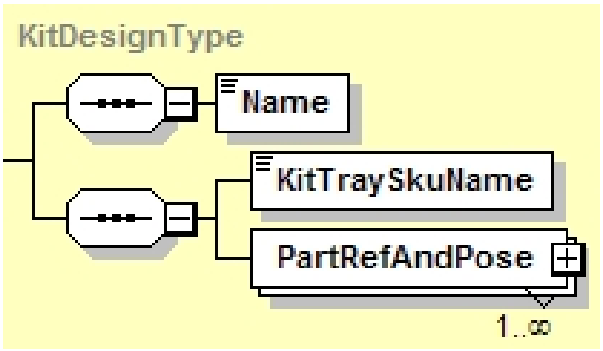

Fig. 4. Kit Design Model

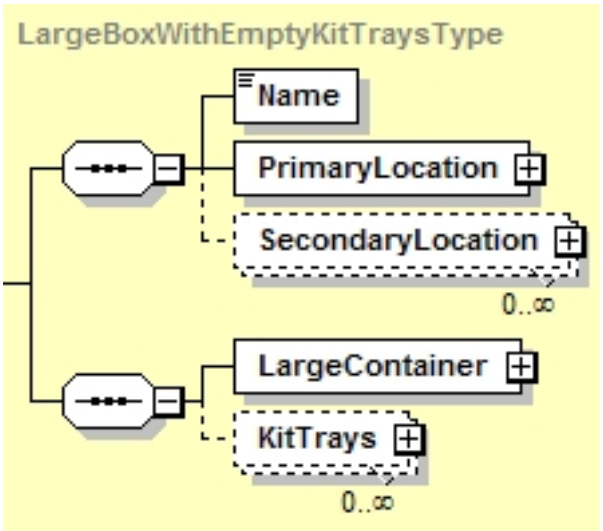

Fig. 5. Large Box With Empty Kit Trays Model

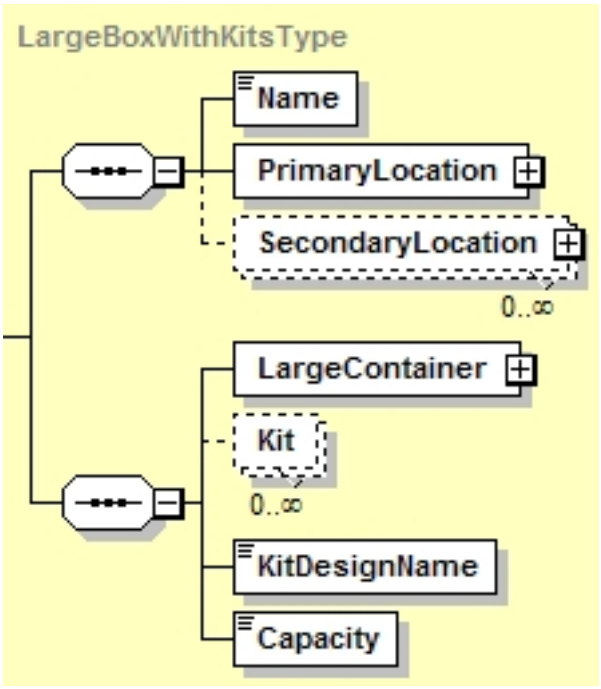

Fig. 6. Large Box With Kits Model

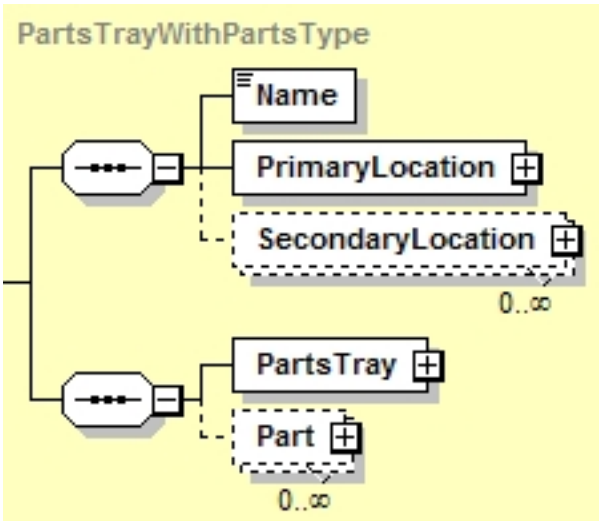

Fig. 7. Parts Tray With Parts Model

- 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 at 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

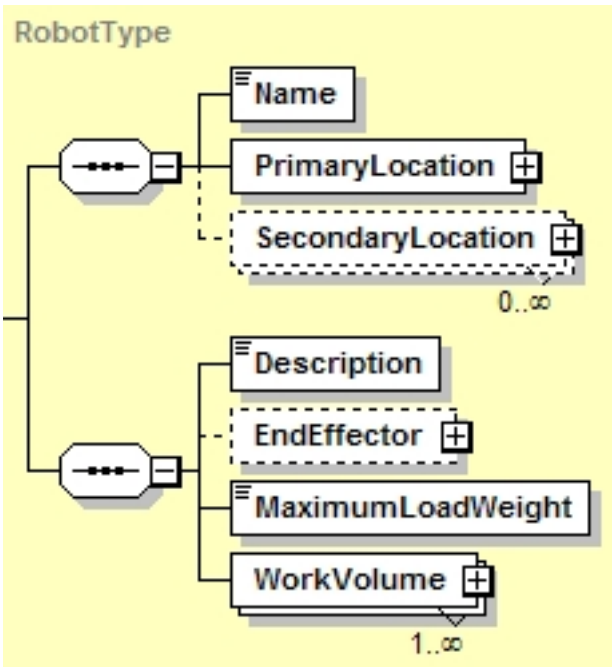

Fig. 8. Robot Model

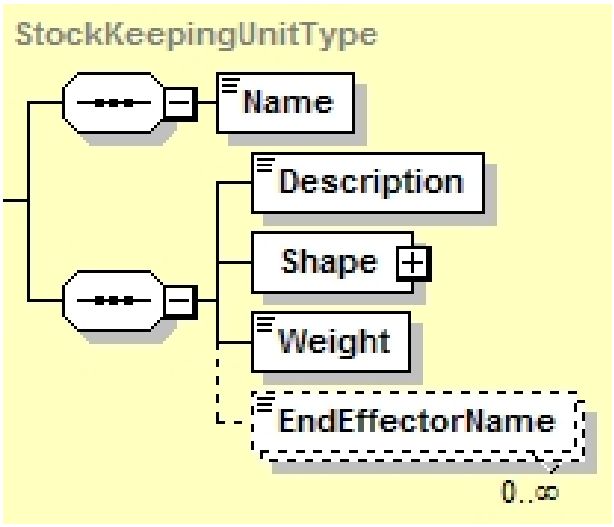

Fig. 9. Stock Keeping Unit Model

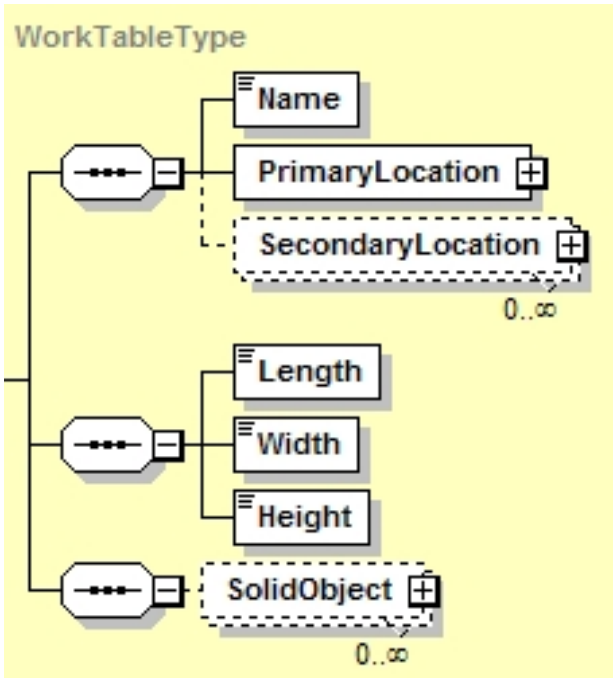

Fig. 10. Work Table Model

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 center 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).

The 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. One or more canonical robot commands may be placed on a queue and executed (in order) when desired. Several additional assumptions are made about the execution behavior of the robot controller. These include:

- 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 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 default coordinate system for poses used in the canonical robot commands is the workstation coordinate system. This may be changed through the use of a CRCL command to be either the workstation coordinate system, the robot base coordinate system, or the tool-tip coordinate system.

The exact syntax of the CRCL commands is provided in Section IV.

III. Implementation

In order to maintain compatibility with the IEEE working group, the ontology has been fully defined in OWL. However, due to several difficulties defined below, the ontology was also fully defined in the XML schema language. 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.
- As shown in Figure 2, 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.
- Attribute names in OWL have a prefix, as described below. The prefixes are not used in XML schema.

A. OWL Specifics

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++ and Java 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 Protégé [16]. Protégé was used for checking the kitting model and data files as they were built. Protégé continues to be used for checking the model and data files whenever they are changed. The layout of the hierarchy in Figure 1 is identical to what may be seen in Protégé's class hierarchy window when the kitting model is loaded.

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 [17], 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. Protégé 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. Protégé 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 Protégé (or any other OWL tool). For example, suppose the line DataPropertyDomain(:hasPoint_X :Point) given above is mistyped as DataPropertyDomain(:hasPoint_x :Point). When Protégé loads the file and the reasoner is started, no errors are detected. Protégé assumes that the DataPropertyDomain for hasPoint_X is unknown (that is not an error in OWL and Protégé) and that there is a new property named hasPoint_x about which the only thing known is its DataPropertyDomain (also not an error in OWL and Protégé, 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 Protégé. Similar errors, such as mistyping the name of an individual, are similarly accepted without error in OWL and Protégé, with similar effects. The difficulties caused by the open world assumption would not occur if Protégé 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. Protégé does not provide any method of specifying that a file is a model file or a data file. The conceptual difference is simple. 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.

B. XML Specifics

To better explore the pros and cons of various representations, 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).

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 elements 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 echoes any XML data file corresponding to the schema. The GenXMiller is still under development and currently handles only a subset of the XML schema language. The GenXMiller is not a new type of system. Several other code generators that use an XML schema as input have been developed [18], [19]. 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 converting XML representations to OWL representations.

The xmlSchemaParser and the GenXMiller use the same underlying parser, which is built in YACC and Lex [20].

In addition to using the xmlSchemaParser and the GenXMiller, a commercial XML tool named XMLSpy [21] 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. If the kitting model is used to represent various starting and goal configurations, 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 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.

(i) Three XML data files were written manually containing the same information as three OWL data files. Each of the OWL files was at least 1,100 lines (20 pages) long. Among the three there were statements of almost all of the types possible under the kittingClasses.owl model. It was decided, therefore, that successful performance for these three files would be an adequate test.

(ii) The three XML data files were run through the owlPrinter to produce three OWL files.

(iii) Since the owlPrinter has a different approach to ordering OWL statements than 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.

(iv) 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.

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 Protégé and no errors were reported.

OWL data files may now be prepared with much less likelihood of human 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 XMLSpy 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 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 anticipated. The approach being using 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.

IV. Canonical Robot Control Language

It is desirable that numerous commercial robot systems be able to immediately execute the plan for the series of actions required to transition from the initial state to the goal state of the kitting problem. However, there is currently no accepted standard robot programming language. For this reason, the authors have developed a canonical robot control language that attempts to be a lowest common denominator of robot programing languages. It is anticipated that kitting plans can be translated into CRCL command sets which may then be evaluated by standardized metric software. The CRCL command sets may then be translated into a specific robot platform's language.

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. 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 in the units of the current operating coordinate system. 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 current operating 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 current operating coordinates of the Z and X axes of the coordinate system of the controlled object.

The complete list of CRCL commands follows.

- 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 units, angle units, and operating coordinate system 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 UnitName 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”.
- SetCoordinateFrame(string CoordSystem) – set the operating coordinate system to the system referred to by CoordSystem. The CoordSystem must be one of “Workstation”, “RobotBase”, or “ToolTip”.
- 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. This applies to the X-axis direction and the Z-axis direction.
- 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 UnitName 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.
- 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.

```
InitCanon()
SetLengthUnits(”meter”)
CloseGripper()
CloseToolChanger()
Dwell(1.7)
Message(”This message is false”)
SetRelativeSpeed(50.0)
SetAbsoluteSpeed(3.8)
MoveThroughTo({{{5,0,2}, {0,0,1}, {1,0,0}},
    {{5,8,2}, {0,0,1}, {1,0,0}},
    {{7,8,2}, {0,0,1}, {1,0,0}}}, 3)
MoveStraightTo({{4,8,2}, {0,0,1}, {1,0,0}})
MoveTo({{9,8,2}, {0,0,1}, {1,0,0}})
OpenGripper()
OpenToolChanger()
SetAbsoluteAcceleration(0.95)
SetAngleUnits(”degree”)
SetEndAngleTolerance(1.3)
SetEndPointTolerance(0.4)
SetIntermediatePointTolerance(10.734)
SetLengthUnits(”mm”)
SetRelativeAcceleration(0.8)
SetRelativeSpeed(0.75)
SetRelativeAcceleration(-110)
MoveStraightTo(87)
EndCanon(2)
```

Fig. 11. Kitting Plan for Testing

A file format for representing CRCL commands has been devised. Figure 11 shows an example of a file prepared using this format. A C++ class model of CRCL commands has been built, and a parser has been built in C++ for reading CRCL files and populating CRCL class instances.

A. Plan Model

The kitting system presented in this document relies on a *direct* model of execution where the executor directly performs the activities specified in the plan. Figure 12 depicts the executor process for the kitting domain where ellipses represent files, regular rectangles are used to define processes, and rounded rectangles illustrate tools. The red dashed box contains the processes part of the executor. The components in Figure 12 are described below:

1) PDDL domain and problem files are currently generated by hand from the IEEE RAS Ontologies for Robotics and Automation Working Group's OWL-based knowledge representation and are used by an open source planner from Coles et al. [22] to automatically generate a plan file. In the near future, these files will be automatically generated. The plan file contains a sequence of actions that can be executed from the initial state and that lead to a goal state. The actions present in the plan file are originally defined in the PDDL domain file. The initial and goal states are defined in the PDDL problem file.
2) The interpreter takes the plan file as input and builds the corresponding CRCL file. Real time information on the environment is required in order to fill in information required by the CRCL on object locations. Since both the OWL and XML implementations of the knowledge representation are file based, real time information proved to be problematic. In order to solve this problem, an automatically generated MySQL database [23] has been introduced as part of the knowledge representation. More details on this database are provided in Section V. Table I shows an example of CRCL commands generated for the PDDL action take-part(part_b_1). Please note that the PDDL action take-part developed for the current kitting domain has more than one parameter. Not all the parameters are relevant for the example depicted in Table I and the number of parameters has been reduced for simplicity. In this example, the locations of the “MoveTo” commands would come from the MySQL database.

take-part(part_b_1)
`Message (``take part part_b_1")`
`MoveTo({{-0.03, 1.62, -0.25}, {0, 0, 1}, {1, 0, 0}})`
`Dwell (0.05)`
`MoveTo({{-0.03, 1.62, 0.1325}, {0, 0, 1}, {1, 0, 0}})`
`CloseGripper ()`
`MoveTo({{-0.03, 1.62, -0.25}, {0, 0, 1}, {1, 0, 0}})`
`Dwell (0.05)`

TABLE I
AN EXAMPLE OF CRCL COMMANDS FOR A PDDL ACTION

3) The CRCL file is used by the controller to create ROS commands.
4) The ROS commands are used by the ROS software controller for a robotic arm to initiate actual execution of actions.

V. MySQL Database for Kitting

While the knowledge representation presented in this paper provides the “slots” necessary for representing dynamic information, the static file structure makes the utilization of these slots awkward. It is desirable to be able to represent the dynamic information in a dynamic database. For this reason, the authors have developed a technique for automatically generating tables for storing, and access functions for obtaining, the data from the ontology in a MySQL database.

Reading data from and to the MySQL database instead of the ontology file offers the community easy access to a live data structure. Furthermore, it is more practical to modify the information stored in a database than if it was stored in an ontology, which in some cases, requires the deletion and re-creation of the whole file. A literature review reveals many efforts and methodologies that have been designed to produce SQL databases from ontologies. Our effort builds upon the work of Astrova et al.[24].

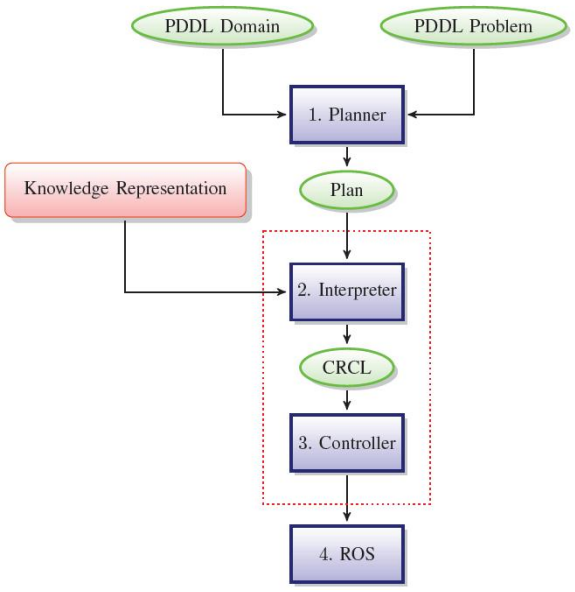

Fig. 12. The executor process

In addition to generating and filling the database tables, the authors have created tools that automatically generate a set of C++ classes for reading and writing information to the kitting MySQL database. The choice of C++ was a team preference and we believe that other object-oriented languages could have been used in this project.

The Generator tool is a graphical user interface developed in Java, allowing the user to store data from OWL files into a MySQL database. This tool also permits the user to query the database using the C++ function calls. The tool Generator is composed of the following functionalities:

1) Convert OWL documents into SQL syntax (OWL to SQL).
2) Translate SQL syntax to OWL language in order to modify an OWL document (SQL to OWL).
3) Convert the OWL language into C++ classes (OWL to C++).

To date, only steps 1. and 3. have been implemented and will be covered in this document. In order to generate the SQL database and C++ classes, the OWL object model must be mapped to the C++ object model and the relational SQL model. To quote the OWL 2 Web Ontology website [13], "Entities are the fundamental building blocks of OWL 2 ontologies, and they define the vocabulary –the named terms– of an ontology. In logic, the set of entities is usually said to constitute the signature of an ontology". Therefore, the notions of single-valued and multi-valued properties as well as the inheritance must be mapped from the ontology to the SQL database and C++ classes. The mapping from OWL proceeds as follows:

- Data properties: In an ontology, data properties link an individual to a data value. Single-valued data properties are mapped into a SQL table entry or C++ class variable with the corresponding type of the original property. For example, in the ontology a robot has a single-valued data property `hasRobot_Description`, represented in the SQL database as a `varchar` and in the corresponding C++ class as `std:string`. Multi-valued data properties are mapped from the ontology into the SQL database as a table and into the C++ class as a `std:vector` with the corresponding type of the original property. For example, in the ontology a stock keeping unit has a multi-valued data property `hasSku_EndEffectorRefs`. This maps to a SQL table containing `varchar` entries and the C++ `std::vector<std::string>` in the corresponding C++ class.
- Object property: In an ontology, object properties link one individual to another individual. The single-valued object properties are mapped to a SQL table entry or C++ class variable. Their type is a pointer to the range of the object properties. For example, in the ontology a solid object has the object property `hasRobot_Description` linking it to a physical location. In the SQL database, we use a foreign key to link the two entries. In the C++ classes, this is represented by a reference to a physical location: `PhysicalLocation* hasSolidObject_PrimaryLocation`. Multi-valued object properties are mapped from the ontology into the SQL database as a table and into the C++ class as a `std:vector` of pointers referencing objects of the range of the property. For example, a solid object also has a list of secondary locations corresponding to a multi-valued object property in the ontology: `std::vector <PhysicalLocation*>hasSolidObject_SecondaryLocation`.

A. MySQL Database Generation

This section provides basic information on the Generator Java tool. Specific information on the tool's usage is included in the tool's manual. Converting an OWL ontology to SQL script files is easily performed using the `Owl to SQL` tab (see Figure 13). The required fields are:

- Ontology Path: The OWL file to be converted. Note that all `Import` statements in this file must use absolute paths.
- Saves Path: The directory where you want to save the SQL files.

Clicking on the "Generate SQL" will generate the SQL script files. Two files will be created by the tool:

- The file used to create tables in the database: `<inputfile>.owlCreateTable.sql`
- The file used to populate the database tables: `<inputfile>.owlInsertInto.sql`.

These files may then be used with the SQL command line interface to create and populate the database.

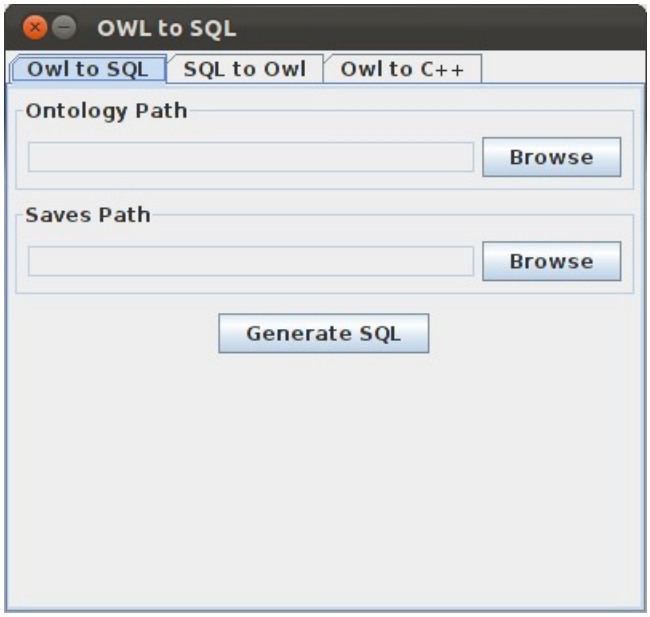

Fig. 13. Owl to SQL tab.

B. C++ Class Generation and Usage

As previously mentioned, the C++ classes are automatically generated by the Generator tool. In addition to the class structure, Data Access Objects (DAO) that are needed to interact with the MySQL database are generated. To map the MySQL database and indirectly the ontology to C++ classes, both the C++ classes and the DAO must be generated.

The C++ class files (.cpp) and header files (.h) are generated in a two step process. The first step does not depend on the content of the ontology, it only initializes the specific objects related to the MySQL connector driver (see Figure 14).

The second step generates all the C++ headers and class files relative to our ontology. All of the `include` statements are made directly in the C++ class files, and only forward declarations are performed in the headers. This resolves problems associated with circular includes or multiple includes. All of the classes include the following methods:

- `get<private field>` - Method for getting a private field.
- `set<private field>` - Method for setting a private field.
- `explode` - Method that splits a string into a vector around matches of a given regular expression.
- `copy` - Method that takes a C++ map as input and copies the values from the map into the instance.
- `get` - Method that reads data from the MySQL database.
- `set` - Method that writes data to the MySQL database.

The actual data access is provided through the use of a data access object (DAO). DAOs provide an abstract interface to some type of database or other persistence mechanism. DAOs map application calls to the database or persistence mechanism, thus providing some specific data operations

```
#ifndef PARTSBIN_H_
#define PARTSBIN_H_
#include <cstdlib>
#include <iostream>
#include <map>
#include <string>
#include <vector>
#include <sstream>

#include "BoxyObject.h"
class DAO;
class PartsBin: public BoxyObject {
    private:
        std::string hasBin_PartQuantity;
        std::string hasBin_PartSkuRef;
        int PartsBinID;
        DAO* dao;
    public:
        PartsBin(std::string name);
        ~PartsBin();
        void get(int id);
        void get(std::string name);
        void set(int id, PartsBin* obj);
        void set(std::string name);
        std::string gethasBin_PartQuantity();
        void sethasBin_PartQuantity(
            std::string _hasBin_PartQuantity);
        std::string gethasBin_PartSkuRef();
        void sethasBin_PartSkuRef(
            std::string _hasBin_PartSkuRef);
        int getPartsBinID();
        DAO* getdao();
        void setdao(DAO* _dao);
        void copy(std::map<std::string,
            std::string> object);
        std::vector<std::string> Explode(
            const std::string & str, char separator);
};
#endif /* PARTSBIN_H_ */
```

Fig. 14. Header of a generated class.

without exposing details of the database. The use of the DAO separates the data accesses that the application needs from how these needs can be satisfied with a specific Database Management System (DBMS), database schema, etc. The different methods of the DAO are the same for any ontology. The concern here is not about the data, but only about the way to retrieve or store it. Only the four vectors filled by the private `fillGetSqlQueries` method differ from one auto-generated C++ file to another.

When the DAO is generated, four vectors are built as follows (shown in Figure 15):

- line 17 : A structure with the SQL query to select the characteristics of an entity. The table relative to the entity itself and the ones relative to its super classes are queried.
- line 18 : A structure with the SQL query to select multi-valued attributes (multi-valued data) for a given entity.
- line 19 : A structure with the names of the tables linked to this entity in the ontology.
- line 20 : A structure with the names of the association tables linked to an object.

With these four structures, one is able to read (`get` method) and write (`set` method) data from and to the MySQL database. The `get` method fills a C++ map and gets the object itself while the `copy` method handles the data. The `set` method is called with a C++ map containing the values of the different attributes as input and writes these values into the MySQL database.

```
#ifndef DAO_H_
#define DAO_H_
#include <cstdlib>
#include <iostream>
#include <map>
#include <vector>
#include <sstream>

#include "Connection.h"
class DAO {
    private:
      std::vector<std::string> className;
      Connection* connection;
      std::vector<std::string> nameDone;
      std::map<std::string, std::string> map;
      std::string path; std::string pathmulti;
      static std::map<std::string, std::string>
        getSqlQueriesDataSingle;
      static std::map<std::string,
                      std::vector<std::string>>
                      getSqlQueriesDataMulti;
      static std::map<std::string,
                      std::vector<std::string>>
                      getSqlQueriesObjectSingle;
      static std::map<std::string,
                      std::vector<std::string>>
                      getSqlQueriesObjectMulti;
      static std::map<std::string,
                      std::vector<std::string>>
                      setSqlQueries;
      static std::map<std::string,
                      std::vector<std::string>>
                      updateSqlQueries;
      void fillGetSqlQueries();
    public:
      DAO(std::string name); ~DAO();
      std::vector<std::string> getclassName();
      void setclassName(
                std::vector<std::string> _className);
      Connection* getconnection();
      void setconnection(Connection* _connection);
      std::map<std::string,std::string>
              get(std::string name);
      void set(std::map<std::string, std::string> data);
      std::vector<std::string> Explode(
                const std::string & str,
      char separator);
};
#endif /* DAO_H_ */
```

Fig. 15. Header of the DAO class.

C. Using the C++ Classes to Access Data from the MySQL Database

Figure 16 depicts an example using the generated classes to retrieve the location of the kit tray `kit_tray_name` from the MySQL database. The different sections of the example are described below:

- lines 1–4: Include the different headers necessary to query MySQL tables. Here, the tables Point, PoseLocation, Vector, and KitTray are required.
- line 9: Initialize an object from the class `KitTray` by passing its name.
- line 10: Allow access to any data from the table KitTray.
- lines 12–13: Initialize an object of type `PoseLocation` and allow access to any data from the table PoseLocation.
- lines 18–19: Retrieve X, Y, and Z coordinates from the table Point for the kit tray `kit_tray_name`.
- lines 22–23: Retrieve the X axis vector (X_i, X_j, X_k) from the table Vector for the kit tray `kit_tray_name`.
- lines 26–27: Retrieve the Y axis vector (Y_i, Y_j, Y_k) from the table Vector for the kit tray `kit_tray_name`.

```
#include "Point.h"
#include "PoseLocation.h"
#include "Vector.h"
#include "KitTray.h"

void CanonicalRobotCommand::
 getKitTrayLocation(string kit_tray_name){

  KitTray* kit_tray = new KitTray(kit_tray_name);
  kit_tray->get(kit_tray_name);

  PoseLocation* kit_tray_pose = new PoseLocation(
  kit_tray->gethasSolidObject_PrimaryLocation()->
                getname());
  kit_tray_pose->get(kit_tray_pose->getname());

  //--Retrieve hasPoseLocation_Point
  Point * kit_tray_point =
  kit_tray_pose->gethasPoseLocation_Point();

  //--Retrieve hasPoseLocation_XAxis
  Vector * kit_tray_x_axis  =
  kit_tray_pose->gethasPoseLocation_XAxis();

  //--Retrieve hasPoseLocation_ZAxis
  Vector * kit_tray_z_axis  =
  kit_tray_pose->gethasPoseLocation_ZAxis();
}
```

Fig. 16. Example using the generated C++ classes.

VI. Test Methods

According to the ASTM International [25, p. vii], a test method is a definitive procedure that produces a test result. It is the authors' desire to develop repeatable test methods that will lead to a better understanding of what it means to have an "agile" and "flexible" planning system and metrics that will allow for the measurement of a system's agility and flexibility. We have chosen to begin our study with the domain of kit building since it is a greatly simplified, but still practically useful manufacturing/assembly domain. However, even in the domain of kit building, a large amount of variance must be accounted for. For example, will parts be rigid or flexible? Will a vacuum effector, or a parallel jaw mechanism, or a fingered gripper be utilized for part picking? Will parts be picked from a tray or a bin? What aspects of the process will be stressed to demonstrate flexibility and agility?

In keeping with the ideas of reduced complexity and repeatability, we will strive to come up with test methods that stress the system's agility and not the system's robotic configuration or abilities. As such, we make the following assumptions:

- All of the contents of the kit will be rigid objects.
- All of the rigid objects are of simple shape (rectilinear) and have a flat surface for a top.
- A vacuum effector is utilized for handling the parts.
- All of the parts will be located in a well-defined parts tray. This will allow for repeatable experiments in terms of part placement.
- All of the part's initial and final positions will be within the reach of the robot.
- There are no obstacles located within the robot's reachable volume.

A. Test Requirements

The test methods themselves are designed as a series of tests that have increasing complexity. It is assumed that the tests will be performed in order, and that a system that is not capable of performing test *n* will fail all subsequent tests as well. For all of the tests, the initial condition of the world and the goal kit configurations are provided as XML and/or OWL files conforming with the IEEE RAS Ontologies for Robotics and Automation Working Group's kitting knowledge representation. The planning system is required to produce an output that will construct the required kit(s) from the initial condition. All of the commands must be in the form of CRCL, and the system is allowed to submit new plans that respond to environmental changes.

The parts supply consists of one or more trays of raw materials and the kit tray is a flat container with separators between locations for individual parts. The part supply trays may be auto-filling (e.g., a single location that is continuously fed with a part) or limited quantity trays. Test methods may be configured to require end-of-tool changes for the picking of various parts.

B. Basic Kit

The first test method is the construction of a single basic kit. The kit will contain two or more different types of parts. The design of the kit and parts supply will be known before the test begins. The actual location of the kit tray and parts supplies will not be known before runtime. The locations and orientations (yaw) of the kit and parts supplies will be varied during consecutive runs of the test method. The planning system is required to submit a single CRCL formatted plan that will construct the given kit. It is assumed that all actions are successful and that there are no execution errors. Each kit construction will be evaluated by our standard metrics as described in Section VII. This test method will evaluate the following aspects of agility and flexibility:

- Ability to correctly build a specific predefined kit.
- Agility in terms of part tray and kit placement. This will show that exact fixturing is not necessary for the construction of a kit.

It should be noted that if the robot is capable of supporting tool changing, different end-of-arm tooling may be required for grasping the various parts required for this test.

C. New Variety of Basic Kit

This test provides the system with a never before seen kit variation. The variation will be delivered in the previously mentioned IEEE RAS OWL/XML format. In addition to the standard kit construction metrics, the time from the receipt of the new kit configuration to the start of first construction will be recorded. The amount of down-time for the cell will also be noted. This test will measure the agility of the robot cell in coping with new kit varieties.

D. Basic Kit, Multiple Varieties

This test builds on the previous test by requiring the construction of two or more kit varieties in a pseudo-random ordering. The kit configurations will be known before the start of the test. This test method will evaluate the following additional aspects of agility and flexibility:

- Ability to correctly build several kits with varying part placements without manual intervention. This will demonstrate agility in terms of kit layout and contents.
- Ability to manipulate items of varying size and weight. This test method will require the workcell to move empty kit trays from storage to a construction location and then to move finished kits to a bin of finished kits.
- If the robot is capable of supporting tool changing, different end-of-arm tooling may be required for kit manipulation.

E. Construction Errors

This test will evaluate the system's ability to recover from predictable error conditions. These conditions will include dropped parts and parts with detected defects. It is expected that the planning systems will interrupt the execution of the kit build in order to provide updated plans to cope with the unexpected events.

VII. Metrics

As described in the previous section, test methods are being developed that will be suitable for comparing the performance of different kitting planning systems. The methods look at plans that are an ordered sequence of actions for a robot to perform. The actions are specified in terms of the CRCL.

A sample plan file is shown in Figure 11. The file is designed for exercising the kittingViewer which is described in Section VIII and is not intended to make sense as a plan. It includes a few intentional errors.

Metrics are being developed that will test both the static performance of the planning system (i.e., end-to-end performance of a single plan without feedback or changes) as well as the execution performance of the system (i.e., the system is allowed to replan due to changes in the environment or action failures). The current metrics were developed with kitting specifically in mind. However, it is envisioned that we will eventually have a taxonomy of metrics where high-level metrics build upon lower level metrics and branches of the taxonomy may be applicable across multiple domains. For many applications, it will be useful to have a single numeric score that represents a system's performance with respect to the individual metrics. This may be accomplished by having a user specify whether each weight is multiplicative or accumulative and specifying weights for each of the accumulative metrics. The metric scores are then multiplied by these weights and combined to form a single score that may be used for comparison.

A. Static Kitting Viewer Metrics

The current metric taxonomy is shown in Figure 17 and is described below. The metrics are designed to be evaluated at the end of each CRCL command with cumulative values.

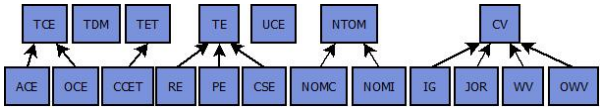

Fig. 17. Kitting static metric taxonomy with abbreviations defined below

- Action Commands Executed (ACE) – the number of action commands that have been executed so far. An action command is any command that takes time to execute.
- Command Sequence Errors (CSE) – the number of commands that are out of sequence. An InitCanon command is out of sequence if it is not the first command in the file. An EndCanon command is out of sequence if it is not the last command in the file. Other commands are out of sequence if they occur before InitCanon or after EndCanon.
- Constraint Violations (CV) – the total of IG, JOR, WV and OWV.
- Current Command Execution Time (CCET) – the time that the current command took to execute.
- Incorrect Gripper (IG) – the wrong gripper was used to pick up an object.
- Joint Out of Range (JOR) – a joint of the robot was commanded to move to an out-of-limit position.
- Number of Objects Moved Correctly (NOMC) – the number of objects that were moved correctly from the parts supply to the kit.
- Number of Objects Moved Incorrectly (NOMI) – the number of objects that were moved to an incorrect position in the kit.
- Number of Total Objects Moved (NTOM) – the sum of NOMC and NOMI.
- Other Commands Executed (OCE) – the number of commands that are not action commands that have been executed so far – mostly setting commands. Executing these commands is assumed to take a negligible amount of time.
- Outside Work Volume (OWV) – the robot was asked to move outside its work volume.
- Parse Errors (PE) – the number of lines in the CRCL command file that cause an error in the command file parser.
- Range Errors (RE) – the number of times a command tries to set a a parameter to a value that is out of the allowed range of the parameter.
- Total Commands Executed (TCE) – the sum of ACE and OCE.
- Total Distance Moved (TDM) – the total distance that the tool tip has moved. This is calculated as the total of the distances between points in the move commands, taken in order (and starting at the place where the controlled point is located initially). The value is updated as each point is reached, not continuously.
- Total Execution Time (TET) – the total time taken so far by executing action commands. This does not include any time that may elapse between when one command finishes execution and when the user tells the system to execute another command. The total execution time is meant to be very close to the actual amount of time that would be taken by the system without user intervention.
- Total Errors (TE) – the sum of the range errors, parse errors, and command sequence errors.
- Useless Command Executed (UCE) – the number of commands that do not change the state of the workstation. Such commands have no effect, so they are useless. The CloseGripper() and CloseToolChanger() commands in Figure 11 are useless because the gripper and tool changer are closed in the initial conditions.
- Weight Violation (WV) – the robot was asked to move some object that violates its load capacity.

B. Execution Metrics for Kit Building

During execution, automated kitting fails to reach its full potential when the supply chain fails and parts and components are not available for kit construction, or when a kit is not properly filled. Part availability failures can be triggered by inaccurate information about the location of the part or part shortage due to delays in internal logistics. Kit construction errors may be due to problems such as improper equipment setup, improper equipment maintenance, part damage, wrong type of part, or part dropped by the robot.

Models for detecting and recovering from plan execution failures mostly deal with *precondition failures*, *action failures*, and *unattributable failures* [26]. Precondition failures appear when all the preconditions for an action are not met during the execution of the action. Action failures are encountered when the execution of an action does not attain its intended effects. Unattributable failures occur when unexpected events caused by external agents change the environment, thus causing the current plan to become obsolete. NIST's Knowledge Driven Planning and Modeling

project has not yet addressed the creation of a taxonomy of execution errors. However, some preliminary metrics have been developed and are described below.

- Manipulation robustness – quantitative and qualitative functionality metrics that describe how well a robot can handle complex objects in complex environments without failing or requiring additional operator interventions. Failures can occur during object detection (lightning variation, shadows), object approach (partially buried), and object manipulation (fragile) operations. It is envisioned that a taxonomy of robustness will be developed that decomposes robustness into areas such as situational awareness, robot accuracy, and grasping dexterity.

- Transporting components – qualitative metrics that describe how well a robotic arm can grasp objects and move these objects from an initial position to a goal position without dropping them.

- Plan generation – as mentioned previously, failures are possible in kitting during the execution of CRCL commands, causing the current plan to become obsolete. In some cases, the current state of the environment is brought back to the state prior to the failure and the robot starts from a "stable" state. In other cases, a completely new plan must be generated by the planning system where the robot starts all over. This metric will measure the planning system's ability to adapt to failures.

- Contact errors – quantitative metrics that keep track of the number of collisions between the robotic arm and objects in the environment. The performance of the robotic arm during kit building is affected by the positions of joints and the end-effector in the environment. The position of the end-effector can reduce the time to complete tasks but can also increase the number of collisions due to joint contact with other objects in a confined space.

- Failures during kit building– quantitative metrics that report the total number of failures encountered during kit building. When a failure occurs during the building of a kit, the number of failures is increased by one. The system may generate a new plan to recover from the failure. If a failure occurs during the execution of the new plan, the number of failures in increased by one again.

- Failure modes recovery – quantitative metrics that represent the number of failure modes the system recovers from through the use of contingency plans. When a failure is detected in the execution process, failure monitors encode appropriate responses (contingency plans) to failure modes for this particular failure.

VIII. Kitting Viewer

A software tool named the "kittingViewer" is being developed that will read files describing the initial state, the goal state, and the plan for getting from the initial state to the goal state. The kittingViewer will simulate execution of the plan, display a view of the plan being executed, and produce and display metrics about the plan. All of the metrics will be numbers. All but one of the metrics will be objective and require no human judgement. The final metric will be a subjective combination of the other metrics in which the other metrics will be weighted and combined as desired by the user.

The kittingViewer is partially built. It is able to read in the three input files and simulate execution of the plan file. Plan metrics are calculated, and robot motion is animated at speeds specified in the plan.

Figure 18 shows the kittingViewer display in its current state of development. The display uses three windows, labeled Metrics & Settings, Kitting Viewer, and Kitting Command. The windows may be moved and resized independently, like other windows in a typical windowing system.

The Kitting Viewer window shows a view of the kitting workstation. The floor of the workstation is covered with a grid. The spacing of the grid is the last entry in the Metrics & Settings window. The robot in the workstation is represented by a gantry robot spanning the entire width of the workstation. The gantry robot moves when any of the CRCL motion commands is executed. The speed at which the picture of the robot is animated matches the actual commanded speed of the robot. When development of the kittingViewer is complete, objects in the workstation will also be shown (in color) and will move if the robot moves them.

The Kitting Command window shows the currently executing command or the most recently executed command, if no command is currently executing.

The Metrics & Settings window shows 9 metrics at the top and 14 settings below that. All but three of the settings correspond to items that may be set using CRCL commands. The extra three are the grid spacing and the robot's maximum speed and maximum acceleration (which may not be reset). As commands are executed, metrics and settings are updated in the window. When the kittingViewer is completed, there will be more metrics.

A. Errors

In order to fully evaluate an input file, many planning errors are noted, but ignored. If a range error occurs, an error message is printed in the terminal window from which the kittingViewer was started. The SetRelativeAcceleration(-110) command in Figure 11 causes two range errors, one because it is negative, and one because its absolute value is greater than 100.

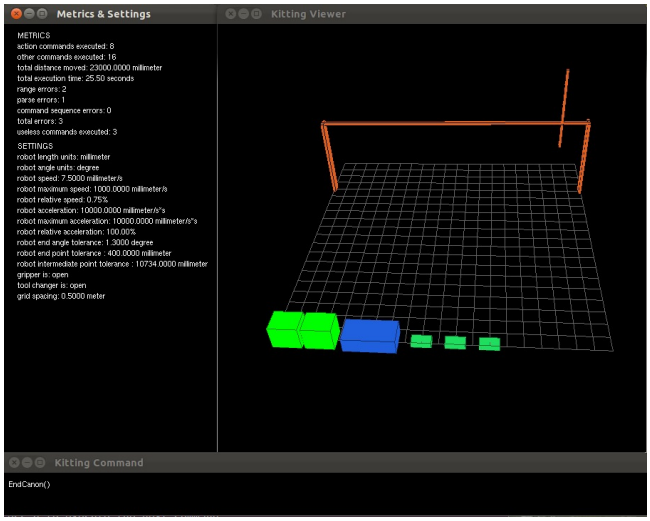

Fig. 18. Kitting Viewer Display

When the parser encounters a line that it cannot parse, it adds an UnreadableMsg to the list of commands it has parsed. The UnreadableMsg includes the text of the line on which the parse error occurred. When the UnreadableMsg is executed, the value of parse errors is increased by one and the UnreadableMsg is displayed in the Kitting Command window so the user can see the line that caused the problem. The MoveStraightTo(87) command in Figure 11 causes a parse error.

B. *Controlling The Kitting Viewer*

Controlling the kittingViewer is accomplished by using the mouse and single keys on the keyboard. When the kittingViewer starts up, a set of one-line instructions is printed in the terminal window from which the kittingViewer was started. Those instructions have the same meaning as the longer explanations given below.

- 'r' key – the 'r' key toggles the behavior of the left mouse button between translating and rotating the picture. This functionality is included because some mice do not have a middle button. Press R once and the left mouse button controls rotation. Press R again and the left mouse button controls translation (the original setting).

- Left mouse button – by default, the left mouse button is used to translate the picture. Position the cursor anywhere in the Kitting Viewer window, hold down the left mouse button and move the cursor by moving the mouse. The picture will move as though it is glued to the cursor. If the R key has switched the left mouse button to rotation, it behaves like the middle mouse button, as described in the next paragraph.

- Middle mouse button – the middle mouse button is used to rotate the picture. This is a little trickier than the other two mouse buttons. To rotate the picture, position the cursor inside the window near an edge of the window, hold down the middle mouse button, and move the cursor in a straight line towards the opposite edge. The picture rotates around an axis that is perpendicular to the line of mouse motion. In order to allow for fine positioning, the amount of rotation that occurs for a given amount of mouse motion decreases as the picture is zoomed in. Hence, if you want to turn the picture completely over, it is best to zoom out, rotate, and zoom back in again.

- Right mouse button – the right mouse button is used to zoom in or out. To zoom out, position the cursor near the bottom of the picture, hold down the right mouse button and push the mouse away from you (moving the cursor up); that appears to push the picture away from you. To zoom in, position the cursor near the top of the picture, hold down the right mouse button and pull the mouse toward you (moving the cursor down); that appears to pull the picture toward you. Moving the mouse side to side while holding down the right mouse button does nothing. There are limits to how far in or out you can zoom. At the highest magnification, it is easy to see a separation of half a millimeter. This is zooming, not moving the point of view, so the eye never goes through the picture.

- 'h' key – if the 'h' key is pressed, the view in the Kitting Viewer window returns to its original position.

- 'g' key – if the 'g' key is pressed when the plan is not completely executed and no action command is executing, the next command in the plan is executed and the Metrics & Settings window is updated. If the g key is pressed when the plan is completely executed or when an action command is in progress, nothing happens.

- 't' key – if the 't' key is pressed, a combined image of all the windows will be saved in a file. The name of the file will be anaglyph_N.ppm, where N starts at 0000 and increases by 1 each time the t key is pressed. The ppm (portable pixmap) format is a common graphics format that many graphics utilities can handle.

- 'z' or 'q' key – if the 'z' or 'q' key is pressed, the kittingViewer program exits, and the windows disappear.

IX. CONCLUSIONS AND FUTURE WORK

The Knowledge Driven Planning and Modeling project is scheduled to continue for an additional two years. During this time, we hope to improve on all aspects of the knowledge

representation and standardization effort. These improvements include increased outreach to industry, improvement of test methods and metrics, and improvements of our ontology and knowledge representation that will be fed to the IEEE working group.

A. Kitting Viewer Development Plans

As mentioned earlier, the kittingViewer is far from complete. We plan to add the following capabilities.

- Add drawing the kitting workstation in its current state. The initial state of the workstation is already available as soon as the XML data file that describes it is read in.
- Add updating the positions of objects as the robot executes commands. It will be necessary to compare the position of the robot with the positions of objects when OpenGripper and CloseGripper commands are executed in order to determine if the robot is grasping them.
- Add metrics related to the positions of objects. This might include (1) the number of objects that should have been moved, (2) the number of objects that were moved, (3) the number of objects that were moved to the correct place, (4) the number of objects that were moved to the wrong place.
- Add metrics related to constraint violations. These might include (1) the number of instances of picking up an object that weighs more than the robot's load capacity, (2) the number of instances of asking the robot to move outside of its work volume, (3) the number of instances of using a gripper to move an object when the gripper is not qualified to move the object. It will also be necessary to decide what the simulation should do in these cases and implement that.
- Add a total score metric and implement finding the total score using a configuration file in which the user assigns weights to the other metrics.

B. Knowledge Representation Development Plans

We have created a knowledge driven system that is capable of building kits in a flexible and agile manner assuming perfect actions. For this system to be practical, this restriction must be removed. To enable this, our current work on the development of a taxonomy of predicates for the situational awareness necessary for kit building will be continued and expanded. The system will also be augmented to allow for the checking of necessary preconditions before actions are executed, and the verification of results after an action has occurred.

To date, we have developed a knowledge representation that supports kitting operations. In cooperation with the IEEE Working Group, this representation will be expanded to support general assembly operations. In addition, we will work with the IEEE Working Group, academia, and industry to standardize the knowledge representations, test methods, and metrics.

REFERENCES

[1] Bozer, Y. A., and McGinnis, L. F., 1992. "Kitting versus line stocking: A conceptual framework and descriptive model". *International Journal of Production Economics,* ***28***, pp. 1–19.

[2] Schyja, A., Hypki, A., and Kuhlenkotter, B., 2012. "A modular and extensible framework for real and virtual bin-picking environments". In Robotics and Automation (ICRA), 2012 IEEE International Conference on, pp. 5246 –5251.

[3] Carlsson, O., and Hensvold, B., 2008. "Kitting in a High Variation Assembly Line". Master's thesis, Luleå University of Technology.

[4] Medbo, L., 2003. "Assembly Work Execution and Materials Kit Functionality in Parallel Flow Assembly Systems". *International Journal of Production Economics Journal of Industrial Ergonomics,* ***31***, pp. 263–281.

[5] Schwind, G., 1992. "How Storage Systems Keep Kits Moving". *Material Handling Engineering,* ***47***(12), pp. 43–45.

[6] Jiao, J., Tseng, M. M., Ma, Q., and Zou, Y., 2000. "Generic Bill-of-Materials-and-Operations for High-Variety Production Management". *Concurrent Engineering: Research and Applications,* ***8***(4), December, pp. 297–321.

[7] Stroustrup, B., 2000. *The C++ Programming Language*. Addison Wesley, New York, NY, USA.

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

[9] W3C, 2004. "XML Schema Part 0: Primer Second Edition". In http://www.w3.org/TR/xmlschema-0/.

[10] W3C, 2004. "XML Schema Part 1: Structures Second Edition". In http://www.w3.org/TR/xmlschema-1/.

[11] W3C, 2012. "OWL 2 Web Ontology Language Document Overview". In http://www.w3.org/TR/owl-overview/.

[12] W3C, 2009. "OWL 2 Web Ontology Language Primer". In http://www.w3.org/TR/owl2-primer/.

[13] W3C, 2009. "OWL 2 Web Ontology Language Structural Specification and Functional Syntax". In http://www.w3.org/TR/owl2-syntax/.

[14] Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins, D., 1998. PDDL–The Planning Domain Definition Language. Tech. Rep. CVC TR98-003/DCS TR-1165, Yale.

[15] Garage, W., 2012. "Robot Operating System (ROS)". In http://www.willowgarage.com/pages/software/ros-platform.

[16] Horridge, M., 2011. *A Practical Guide To Building OWL Ontologies Using Protege 4 and CO-ODE Tools, Edition 1.3*. University of Manchester, Manchester, England.

[17] ISO, 2003. *Industrial automation systems and integration – Product data representation and exchange – Part 11: Description method: The EXPRESS language reference manual*. ISO, Geneva, Switzerland.

[18] Briganti, D., 2012. utilities-online.info, 11.

[19] Apache.org, 2012. Xml schema, 11.

[20] Levine, J. R., Mason, T., and Brown, D., 1995. *lex & yacc*. O'Reilly, Cambridge, MA, USA.

[21] GMBH, A., 2010. *Altova XMLSpy 2010 User & Reference Manual*. Altova GMBH, Vienna, Austria.

[22] Coles, A. J., Coles, A., Fox, M., and Long, D., 2010. "Forward-Chaining Partial-Order Planning". In 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, AAAI 2010, pp. 42–49.
[23] Corporation, O., 2012. Mysql, November.
[24] Astrova, I., Korda, N., and Kalja, A., 2007. "Storing owl ontologies in sql relational databases". *World Academy of Science, Engineering and Technology,* ***29***, pp. 167–172.
[25] ASTM, 2012. Form and style for astm standards. Web, March.
[26] Myers, K. L., 1998. "Towards a Framework for Continuous Planning and Execution". In Proceedings of the AAAI 1998 Fall Symposium on Distributed, AAAI Press.