Functional Requirements of a Model for Kitting Plans

Stephen Balakirsky
NIST MS 8230
100 Bureau Drive
Gaithersburg, MD 20899, USA
301-975-4791
stephen.balakirsky@nist.gov

Zeid Kootbally
Dept. of Mechanical Engineering
University of Maryland
College Park, MD 20742, USA
301-975-3428
zeid.kootbally@nist.gov

Thomas Kramer
Dept. of Mechanical Engineering
Catholic University of America
Washington, DC 20064, USA
301-975-3518
thomas.kramer@nist.gov

Raj Madhavan
Maryland Robotics Center
Inst. for Systems Research, UMD
College Park, MD 20742, USA
301-975-2865
madhavan@umd.edu

Craig Schlenoff
NIST MS 8230
100 Bureau Drive
Gaithersburg, MD 20899, USA
301-975-3456
craig.schlenoff@nist.gov

Michael Shneier
NIST MS 8230
100 Bureau Drive
Gaithersburg, MD 20899, USA
301-975-3421
michael.shneier@nist.gov

NIST = National Institute of Standards and Technology

ABSTRACT

Industrial assembly of manufactured products is often performed by first bringing parts together in a kit and then moving the kit to the assembly area where the parts are used to assemble products. Kitting, the process of building kits, 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 building models of the knowledge that will be required to operate an automated kitting workstation. A first pass has been made at modeling non-executable information about a kitting workstation that will be needed, such as information about a robot, parts, kit designs, grippers, etc. A model (or models) of executable plans for building kits is also needed. The plans will be used by execution systems that control robots and other mechanical devices to build kits. The first steps in building a kitting plan model are to determine what the functional requirements are and what model constructs are needed to enable meeting those requirements. This paper discusses those issues.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *frameworks, data types and structures, classes and objects, control structures.*

General Terms

Design, Standardization, Languages

Keywords

assembly, functional requirements, kitting, language, model, planning, process planning,

PerMIS '12 March 20-22 2012 College Park, MD, USA

1. INTRODUCTION

Industrial assembly of manufactured products is often performed by first bringing parts together in a kit and then moving the kit to the assembly area where the parts are used to assemble products. Kitting, the process of building kits, 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 building models of the knowledge that will be required to operate an automated kitting workstation. A first pass has been made at modeling non-executable information about a kitting workstation, such as information about a robot, parts, kit designs, grippers, etc. The model is written in Web Ontology Language (OWL) [5]. A model (or models) of executable plans for building kits is also needed. Thus far, we have only a mock-up of a sample plan that includes a natural language description of the elements of a plan model. We intend to build that model in OWL, also. The plans will be used by execution systems that control robots and other mechanical devices to build kits. The first steps in building a kitting plan model are to determine what the functional requirements are and what model constructs are needed to support those requirements.

We are working towards developing standard representations of both kitting workstations and process plans for kitting. Kitting is accomplished by discrete processes, so we consider only process plans for discrete processes. We are also committed to using hierarchical control. In the case of a kitting workstation, there are at least two control levels, the workstation level and the robot level. The robot controller will take commands from the workstation controller. In this paper we deal only with planning models for the workstation level.

In section 2 we introduce existing process plan models. Section 3 discusses functional requirements for the language used to build a process plan model. Section 4 presents constructs often found in process plan models, relates them to the functionality they serve, and describes the extent to which we are currently planning on using them in the plan model for kitting. Section 5 discusses planning considerations that affect the need for various types of functionality. Section 6 describes how we plan to evaluate the adequacy of the model for kitting process plans and gives suggestions for further work.

This paper does not deal with process models or with the connection between process plans and processes.

2. EXISTING PROCESS PLAN MODELS

A few standard discrete process plan models and many non-standard models have been developed.

2.1 Standard Process Plan Models

Since we are interested in standards, we have examined existing standards for discrete process plans. These include the following.

ISO 10303, generally known as STEP (STandard for the Exchange of Product model data), includes a Part 49, "Process structure and properties" [1]. The model is built in the EXPRESS language (as are all ISO 10303 models). It is a very general, domain-independent model. The central object of the model is "action_method." Other than describing itself as specifying "the elements of a process plan," which is "the specification of instructions for a task," the document containing the model provides no description of the functionality of a process plan that it is intended to provide. The concepts defined in STEP Part 49 are used in Part 240 of STEP, which is focused on process plans for machined products [3].

Part 10 of ISO 14649 [2] "specifies the process data which is generally needed for NC-programming within all machining technologies." This includes the definition of a general process plan model that might be used outside of the machining domain as well as inside. A central element of the model is "Executable." Instances of Executable "initiate actions on a machine" when executed. Part 10 of ISO 14649 is remodeled in STEP terms in STEP Part 238 [4].

Languages for programming machine tool controllers may be regarded as process plan models. The Dimensional Measuring Interface Specification (DMIS) is the only standard language for writing programs to be executed by the controller of coordinate measuring machines [6]. DMIS has many of the plan constructs described in section 4.

2.2 Other Process Plan Models

Since we are using OWL to model the environment of a kitting workstation, we looked at the Process.owl section of the most recent version of OWL-S [7]. OWL-S was originally developed for the World Wide Web Consortium (W3C), but was never adopted as a W3C recommendation.

There is an enormous body of literature regarding planning, particularly planning on state spaces. The book *Automated Planning* [9], for example, includes over 500 references. There is a correspondingly wide variety of plan models. We do not attempt to describe them here.

The language *A Language for Process Specification* (ALPS) was developed at the National Institute of Standards and Technology (NIST) and has been used in a few NIST projects [10].

3. LANGUAGE AND STRUCTURE FUNCTIONAL REQUIREMENTS

The language used to represent a process plan model should make it possible to use plan instances easily. Specifically, the model should be written in a language that is automatically processable into computer code that (1) has data structures for representing a plan, (2) can read a plan file and save it in terms of the automatically defined structures, (3) has access functions for getting data out of the structures and putting data into them, and (4) can write a plan file from the structures. Languages for which mature software exists that can generate computer code as just described include EXPRESS and XML schema. Software exists that can process OWL that way, but it is at an early stage of development and is not yet widely used [11].

Even if a code generator for a language can do the four things listed above, if the structure of the model is too general, the information available by using the access functions may be too atomic for even an expert in both programming and process planning to use readily. This is the case with both Part 238 of STEP and Part 240 of STEP. A set of computer code written by a STEP expert is required to extract meaningful process plan data usable by a programmer building a process planner or a process plan executor. On the other hand, automatically generated code built by processing ISO 14649 (which, like the two STEP parts, is written in EXPRESS) using the same code generator is readily usable by an application builder [17].

It is not sufficient, of course, to have a plan model. It must be possible to represent plans that are instances of the model. For EXPRESS models, there is more than one standard way in which this may be done, the most commonly used of which is a "physical file" [11]. For XML schema, instances are built in XML files that conform to the schema [18]. For OWL, instances and structures may be put into the same file. It is more convenient, however, to have a fixed structures file and build instance files that use the structure file via an "import" statement. Curiously, while C++ is a widely used standard programming language [16] and would be entirely adequate for building class models of many planning domains, there is no standard textual data file representation for C++ class instances.

4. PLAN CONSTRUCTS

The plan model needs to be rich enough to represent all aspects of the kit building process. This process includes operations ranging from selecting the appropriate gripper for moving kit trays or parts to iterating through a list of steps that place parts in a kit. In order to meet these requirements, we have examined techniques for representing parameters, variables, resources, and actions.

An abbreviated example kitting plan is shown in Fig. 1 using XML format. The example uses the constructs described in the remainder of this section.

```
<ProcessPlan>
 <About>
  <PlanId>kitABCPlan</PlanId>
  <PlanVersion>1.0</PlanVersion>
  ...
  <TargetSKU>kitABC</TargetSKU>
 </About>
 <PlanRequirements>
  <PlanRequirement>
   <Name>boxOfEmptyTrays</Name>
   <Type>LargeContainer</Type>
   <SkuRef>Box1</SkuRef>
   <ContentsType>
    <SkuRef>KitTrayX</SkuRef>
   </ContentsType>
  </PlanRequirement>
  ...
 </PlanRequirements>
```

```
  <PlanParameters>
  <PlanParameter>
    <Name>NumberOfKitsToMake</Name>
    <Type>positiveInteger</Type>
  </PlanParameter>
 </PlanParameters>
 <InternalVariables>
  <InternalVariable>
    <Name>BoxWithEmptyTrays</Name>
    <Type>LargeContainer</Type>
    <Requirement>boxOfEmptyTrays</Requirement>
  </InternalVariable>
  <InternalVariable>
    <Name>CurrentKitTray</Name>
    <Type>KitTray</Type>
    <SkuRef>KitTrayX</SkuRef>
    <InitialValue>NULL</InitialValue>
  </InternalVariable>
  ...
 </InternalVariables>
 <ToDo>
  <Start></Start>
  <DoInGivenOrder>
    <DoInAnyOrder>
      <Bind>
        <Variable>BoxWithEmptyTrays</Variable>
        <WhichOne>ANY</WhichOne>
        <ErrorAction>QUIT</ErrorAction>
      </Bind>
      <Bind>
        <Variable>BoxForFullTrays</Variable>
        <WhichOne>ANY</WhichOne>
        <ErrorAction>QUIT</ErrorAction>
      </Bind>
      ...
    </DoInAnyOrder>
    <Set>
      <Variable>n</Variable>
      <Value>0<Value>
    </Set>
    <LoopInGivenOrderWhile>
      <Test>n LessThan NumberOfKitsToMake</Test>
      ...
      <RobotMoveAbove>CurTrayPose</RobotMoveAbove>
      <RobotPickUp>CurrentKitTray</RobotPickUp>
      <RobotMoveAbove>KitTrayPose</RobotMoveAbove>
      <RobotPutDown>
        <What>CurrentKitTray</What>
        <Where>KitTrayPose</Where>
      </RobotPutDown>
      ...
      <Set>
        <Variable>n</Variable>
        <Value>n+1<Value>
      </Set>
    </LoopInGivenOrderWhile>
  </DoInGivenOrder>
  <Stop></Stop>
 </ToDo>
</ProcessPlan>
```

Fig. 1 Kitting Process Plan Example (abbreviated)

4.1 Plan Parameters

Plans can have parameters, such as the name of a file of decision rules to use or the number of kits to be put together. If plan parameters are used, structures to support their use may be needed in the plan model. Having a parameter for the number of kits, for example, requires some structure that implements looping. Plan parameters are set in the command to execute the plan. Typically, plan parameters are not reset during plan execution. Plan parameters serve the function of allowing execution time specification of what to do or how to do it. The current kitting process plan model has a plan parameters section.

4.2 Plan Variables

Plan variables are variables set in the course of executing a plan, not in the command to run the plan. It is useful if plan variables have specific data types. A given variable may represent different objects of the same type during plan execution. The current kitting process plan model has an InternalVariables section that contains plan variables.

4.3 Resources

The current kitting process plan model has a PlanRequirements section that gives required resources.

4.3.1 Resource Requirements

A process plan that is intended to be executable should make it easy for a user to determine if the resources required to execute the plan are available. The straightforward way to do this is to have a separate section of the plan that lists the required resources. Each step of a plan should identify each resource it requires beyond what the plan as a whole assumes is available. It is not sufficient, however, to mention resources only as they are associated with steps of the plan, since if only that is done, it may be difficult to determine the total set of required resources.

A plan model for a specific domain (a kitting workstation, for example) may assume the availability of fixed resources in the environment (a robot, for example). The resource section of a plan does not need to include those resources. If a plan is intended to be usable in several different environments of the same type (different kitting workstations, for example), then the resources section of the plan will need to include specific values applicable to the fixed resources in that type of environment (the extent of a robot work volume, for example).

Where plans include alternative actions and those actions use different resources, it may be hard for the user to determine if available resources are adequate. Where one resource may be substituted for another and at least one of a set of alternative resources must be used, there is no difficulty. The list of required resources simply contains sets of mutually substitutable resources (three alternative grippers, for example). If alternative ways of executing the plan require different sets of resources, there is a problem. On the one hand, it is counterproductive to force the user to assemble all the resources that might be required. The user should have to assemble only a minimal set of required resources. On the other hand, until an execution of the plan is performed, it is not known which resources will be used. Where decisions on which alternative to use are made on the basis of environmental conditions that change slowly, one way to deal with this is to run a simulation of executing the plan. Then the resource requirements can be pared down to those resources used in the simulation. At the same time, the plan would be pruned of those branches that are not used. The reduced plan would be usable as

long as the conditions under which the reduced plan is executed are close enough to those under which the reduced plan was generated. Simulation before execution is also useful when a user has a set of resources and a complex plan and needs to determine if the plan can be executed with that set of resources.

4.3.2 Resource Descriptions

The description of a resource might be given at any of three levels of abstraction.

- a description of the capabilities of the resource (for example, the lifting capacity and maximum opening of a gripper)
- a specification of a resource in a catalog (for example, GripCo model 123)
- a specific instance of a resource (for example, GripCo model 123 with serial number ABC).

Which resource description level to use, if any, depends on the level of abstraction a plan is intended to have. Section 5 discusses levels of abstraction.

4.4 Actions

The actions section of a plan specifies what to do. This is by definition a functionality every sort of process plan must have. The actions section must always include tasks. The actions section may also include explicit control structures, or information to be used for control may be contained in the task description. In any event, some method of controlling the order in which tasks are performed is required. The current kitting process plan model has a ToDo section that contains the actions.

4.4.1 Explicit Control Structures

The current kitting process plan model includes all of the following types of control structure except for synchronous operation, DoSimultaneously, and DoSome.

4.4.1.1 Do In Given Order

In many process planning models and most computer programming languages, the default rule for execution order is to do things in the order in which they are listed in the file, and there is no explicit control structure for doing things in that order.

The functionality of being able to execute plan steps in the order in which they are given in a file is very convenient. Unless a process plan model uses implicit control structures throughout the actions section, the model should include a default rule or an explicit DoInGivenOrder control structure. If the plan model includes explicit commands for ordering, such as described immediately below, then having an explicit DoInGivenOrder will help avoid confusion.

4.4.1.2 Do In Any Order

In theory, an extraordinarily simple process plan language might specify in its natural language execution rules that the steps of all plans may be executed in any order. In any realistic plan model, however, if the ability to say that some set of steps may be performed in any order is needed, then an explicit control structure implementing this functionality is needed.

A DoInAnyOrder functionality is desirable if it is expected that there will be circumstances in which no particular task order is required and the system executing the plan is either capable of multitasking or is expected to have better information available for setting the order than is available at the time the plan is made.

The DoInAnyOrder control structure might have subtypes that allow or disallow simultaneous execution of tasks. If the execution system is known be able to perform operations in parallel, then the plan model should include a DoSimultaneously control structure that requires parallel execution.

4.4.1.3 Do One

The DoOne control structure is followed by a list of alternatives. The execution system picks one alternative and executes it. The execution system is free to pick any of the alternatives. It may pick one at random, or it may evaluate the goodness of the alternatives by whatever criteria it prefers and pick the best one. The alternatives will usually have the same primary effect but may have different secondary effects. For example, in kitting, if it is necessary to get at box A which is underneath box B, the plan might include a DoOne with the alternatives of putting box A on the table or putting box A on box C.

Some languages include a DoSome control structure that specifies that any N of a set of alternatives should be executed. This is more powerful than DoOne, since when N is 1, it is equivalent to DoOne, but occasions when N is not 1 will probably be rare – remove three of the six boxes on the table, for example.

4.4.1.4 Branch on Condition

Another type of control structure includes a condition to be tested followed by a specification of what to do if the condition is met. In common computer languages, these are called if or switch or select. All of them may be combined with else, which specifies what to do if none of the explicit conditions is met. Switch and select have cases. Implementing condition testing requires that the plan language include variables and (usually) expressions, for example, "(x+y) > 3" is a condition that is a Boolean expression using a less than operator to compare an arithmetic expression containing variables and an addition operator with a numerical constant. The Boolean test may be implicit rather than explicit, but variables are always needed.

Branching on a condition is a functionality that is hard to do without whenever a plan model includes plan parameters and/or variables.

4.4.1.5 Loop

When a set of steps must be repeated a number of times or as long as a condition holds, a control structure that implements looping (iteration) is needed. The simplest form of loop simply states that a set of steps must be executed N times, and there is no explicit test (the execution system is expected to keep track), but in most of the many varieties of loop structure ([15] has a 40-page chapter on looping), a condition is tested at some point in the loop that stops the looping.

For kitting plans, our model includes LoopInAnyOrderWhile and LoopInGivenOrderWhile. In these control structures, a condition is tested before any step in the list of conditional steps is executed. The rest of the action of these loops is as implied by their names.

4.4.1.6 Synchronous Operation

If two devices must operate together to accomplish something (such as two robot arms picking up opposite ends of a pipe), a control structure for synchronization is needed in the plan.

4.4.1.7 Create and Destroy Instances

Control structures that are able to create and destroy instances will be useful in the plan model for any activity in which instances come into existence or go out of existence during plan execution. In kitting, for example, kits come into existence that did not exist before plan execution was started, and part supplies go out of existence when they are empty (the empty container that remains is no longer a part supply).

4.4.1.8 Bind Resources

A model of a step that binds a plan variable to an instance of a resource is useful in the plan model. When a "bind" step is executed, a plan variable representing a resource is set to a specific object in the workstation matching the description of a resource. This requires being able to obtain information about what is in the workstation. Such information would reside in a dynamic knowledge base, so implementing resource binding requires that a dynamic knowledge base be available to the plan executor. Resource binding might be combined with resource allocation. For example, when a bind command is executed, the resource might be marked as unavailable as the value of a set command or another bind command.

4.4.1.9 Set Variables

A model of a step that sets a plan variable to a value is useful in the plan model. When a "set" step is executed, the value of a plan variable is set. The value to which the variable is set may be obtained by a straightforward knowledge base inquiry (such as the location of a solid object) or it may be obtained by evaluating an expression (for example, (a + b)) or making a function call (for example, a call to a function that returns the first item in a list). The last two methods, of course, require that the plan model include an expression model and a function model.

4.4.1.10 Start and Stop

Because explicit start and stop control structures simplify executing plans, the plan model should include Start and Stop. Only one Start step is allowed in a plan, and it must be the first step. Either multiple Stop steps or only one might be allowed. If only one is allowed, it must be the last step.

4.4.2 Implicit Control Structures

The order in which steps of a plan are executed may be controlled implicitly by putting a list of predecessor (and/or successor) steps into each step. In some implementations of this ([10], for example), only "join" steps, which are steps that join threads coming from a matching "split" step may have more than one predecessor. In other implementations, any step may have multiple predecessors, and the control rule is that all the predecessors of a step must be executed before the step may be executed. The two approaches may be combined using split/join pairs that enable/disable the use of multiple predecessors. This was implemented in [13]. The use of multiple predecessors allows the plan to be executed in multiple orders that would otherwise be allowed only by including a combinatorial explosion of split/join pairs.

4.4.2.1 Do In Precondition Order

Enabling the use of multiple predecessors for a portion of a plan may be implemented by the DoInPreconditionOrder control structure. A DoInPreconditionOrder step is followed by a list of steps, each of which has a sequence number and a list of the sequence numbers of other steps that must be executed previously. All the steps in the list must eventually be executed.

4.4.3 Support Structures

Where steps or conditions in a plan require numbers or Boolean values, it is convenient if plan parameters, plan variables, object properties, operator expressions, and functions are used. These all may be classed as subtypes of expression. Some plan models, such as STEP part 49 and ALPS, observe that an expression model is required without modeling one. Other plan models, such as DMIS, include explicit models of expressions.

For kitting, in order to deal with location information and do geometric reasoning, all of the support structures just listed are required. For example, in order to take a part out of a part supply, a function that finds the first part remaining in the part supply is needed, and the location property of that part must be found in order to generate an instruction telling the robot where to go to pick up the part. As another example, if there is a stack of empty trays in a box and we want to pick up the one on top (which is not necessarily the first one in the list of trays in the box), a function that finds the tray on top is needed.

4.4.4 Kitting Actions

A set of task types specific to kitting is required in a kitting process plan model. The last subsection of this subsection presents the task types we intend to use first. The stage is set by brief descriptions of the objects in a kitting workstation, the scenario our plan model must support, and the execution model we intend to follow.

4.4.4.1 Objects in a Kitting Workstation

Our initial kitting workstation model is relatively simple. A kitting workstation contains some fixed equipment: a robot, a work table, a part gripper, a tray and kit gripper, and a gripper changing station. Items that enter the workstation include empty kit trays, boxes in which to put finished kit trays or empty part supply trays, and part supplies. A part supply may be a tray or box with parts inside in known or unknown locations or a box containing trays with parts. Items that leave the workstation may be boxes with finished kits inside, empty part trays, empty boxes, or boxes with empty part trays inside.

4.4.4.2 Scenario

In our kitting project, the first version of the plan model is designed to support the following scenario. An external agent (which we call the factotum) sets up the workstation by putting into it:

- a box of empty kit trays (may be only partially full)
- a box for finished kits (may have some kits in it already)
- a box for empty part supply trays
- several part supply trays

The knowledge base for the workstation includes descriptions of the designs of kits, parts, and trays involved. The knowledge base also has descriptions of where all the objects in the workstation are. The factotum that sets up the workstation fills in the knowledge base so that it describes the setup correctly. After the initial setup, objects are expected to move only if the robot or factotum moves them. Whenever an object is moved by the robot or factotum, its location is updated. The workstation control system builds kits by:

- telling the robot to take an empty kit tray out of the box of empty kit trays and to put it on the work table
- telling the robot several times to take a part out of a part supply and put it in the kit being built

- telling the robot, whenever a kit is finished, to put the finished kit in the finished kit box
- telling the robot to change its gripper as necessary for handling either parts or part trays
- telling the factotum, whenever necessary, to remove empty parts trays, to put part supplies in, to put boxes of empty kit trays in, or to remove full boxes of finished kits.

4.4.4.3 Kitting Task Execution

The scenario is carried out by having the workstation controller execute a workstation level process plan. The workstation controller can, by itself, execute steps that set variables, choose among alternatives, etc. To move things, however, the workstation controller requires the robot or the factotum to execute instances of specific types of kitting tasks. This is expected to be accomplished by having the workstation controller send a command to the robot controller or the factotum. The robot controller or factotum will carry out the command and report back whether command succeeded or failed. If the command succeeds, the workstation controller will execute the next step in the plan. If the command fails, the workstation controller will either just stop executing the plan or deal with the error condition outside of executing the process plan and then resume executing the plan. As currently envisioned, resuming plan execution after an error will be feasible only if the error condition can be corrected and the workstation environment can be set to the state it would have been in if the command that failed had succeeded.

Currently, the kitting process plan model contains no error handling tasks. The workstation controller is expected to deal with error conditions independently from executing the process plan.

4.4.4.4 Types of Kitting Tasks

The task types that have been defined to enable writing a plan that follows the scenario include the following.

- FactotumRefill - This is followed by a variable representing a requirement. When the statement is executed, the factotum puts an object of the required type in the workstation and updates the workstation model.
- FactotumRemove - This is followed by a variable representing the object to remove. When the statement is executed, the factotum removes the object and updates the workstation model.
- FactotumReplace - This is followed by a variable representing the object to replace. When the statement is executed, the factotum removes the object, puts another object of the same type in the same place, and updates the workstation model. The new object should be different from the old one in an appropriate way.
- RobotChangeEndEffector - This is followed by the name of an EndEffector to change to. When the statement is executed, if the robot is not already holding the named EndEffector, the robot moves to the changing station, puts down the EndEffector it has (if it has one) and picks up the named EndEffector. If the robot is already holding the named EndEffector, no action is taken.
- RobotMoveAbove - This is followed by a Pose. When this statement is executed, the controlled point on the robot's end effector moves to a point that has the same X and Y values of the location of the Pose but has a greater Z value by some amount the executor thinks will be sufficient so that the robot will not collide with anything near the location point. This statement is not particularly well defined and might be modified.
- RobotPickUp - This is followed by a variable whose value is the object to pick up. When the statement is executed, the robot moves its gripper down into position for grasping the object, the gripper grasps the object, and the robot moves up so that the height of the lowest point of the object is the same as what the height of the lowest point of the gripper was previously.
- RobotPutDown - This is followed by a variable representing the object to put down and a variable representing the Pose of the object at which the object should be released. When the statement is executed, the robot moves the object into the given Pose and releases the gripper's grip on the object. Then the robot moves up so that the lowest point of the gripper is clear of the object that was put down.

4.5 Other Plan Contents

A process plan file needs to include information that may be used to keep track of the document. This information is not used by the process plan execution system at execution time, though it may be used immediately before execution starts to verify that the right plan is being used. The current kitting process plan model includes an About section with subsections for PlanId, PlanVersion, PlanDateAndTime, PlanAuthor, PlanWorkstation, Description, and TargetSKU (an identifier for the stock keeping unit data that is a detailed description of the type of kit to be made).

5. PLANNING CONSIDERATIONS

The ways in which plans are intended to be generated and used is a major consideration in deciding what constructs to include in the plan.

5.1 Abstraction

The most abstract (or high-level) plan may specify only the intended effects of the plan. For a kitting workstation, a high-level plan might state that a number of kits of a particular type are to be made. For a quality control system, a high-level plan might state that parts of a particular type are to have the tolerances on a particular set of features checked.

If a plan is intended to be executable, the plan should include resources, executable operations, and whatever degree of ordering is required for executing the operations.

In many industrial settings, it is useful if a process plan can be refined in stages. The NIST Manufacturing Systems Integration (MSI) project, for example, identified three stages, which were called (1) process plans (2) production-managed plans, and (3) production plans [12]. As used in the MSI project, "A production-managed plan is an expansion of a process plan which supports the production of a required number of products using a given factory configuration. A production plan is a refinement of a production-managed plan which identifies specific resources for each step and the times of

their usage for that step." As described, a production plan is a combination of a plan and a schedule. Scheduling, in our view, is beyond the scope of a process plan, but supporting the other types of refinement described by the MSI project, as well as refinement by pruning branches of a plan, is a functionality that may be required of a process plan model. With this functionality, a single plan model will support both a plan and any refinements of it (possibly in a chain of successive refinements). This implies, for example, that all levels of action abstraction and resource description should be supported by the plan model.

Plan refinement was implemented in the Feature Based Inspection and Control System at NIST [13] and was discussed in [14]. Refining a plan may require generating a separate document containing the refined plan. A plan model that supports representing both a plan and its refinement in a single document may be unnecessarily complex. Regardless of the way in which refinement is handled, there must be a link from any refinement back to the plan it refines.

5.2 Decision-making Responsibilities

Planning decisions might be made in either the planner or the plan executor. Depending on the assignment of planning responsibilities to the planner or the plan executor, the functional requirements of the plan model may be very different.

At one extreme, if the planner knows enough to make all the decisions, a plan format may suffice that is simply an ordered list of tasks to perform. In this case, since no decisions need to be made at execution time, no Boolean expressions, if-thens, or structures that allow alternatives are needed in the plan. In addition, since the natural form of a file is an ordered list, no ordering structures are needed. All that is necessary is to be able to tell where one step ends and the next begins. Because a file is an ordered list by nature, the most abstract plans will require using a structure such as DoInAnyOrder that is able to disorder the steps.

At the other extreme, if there may be foreseen but random changes in the environment in which the plan is executed (e.g., the robot is apt to drop things) or if the conditions of the environment are not known at planning time (e.g., the location of the part supply is not set until execution time), the plan will need to include items such as variables, if-thens, sets of alternatives, and Boolean expressions.

5.3 Extendible Generic Plan Model

It is extremely desirable to have a generic model of process plans that may be extended into specific domains. If models for different domains build on a common core, people who understand the plan model for one domain can gain understanding of other plan models much more easily than if there is no common core. Similarly, it will be possible to use the core software of a system that executes plans in one domain when building a plan execution system for a new domain.

Because the target level of plan abstraction varies from application to application, the generic model must be built so as to support different levels of abstraction efficiently and clearly. It may be possible to support different levels of abstraction by using optional elements. This notion needs further examination since items that are optional at a high level may be required at lower levels.

A generic plan model might specify the sections of the plan, control structures, some aspects of resource description, and a generic task. Specializations of the generic plan for specific domains would have specialized resource and task descriptions that are subtypes of generic tasks and resources.

5.4 Human Comprehensibility

With a human in the loop during plan generation (always or as needed), the range of good plans that can be generated expands greatly. Thus, one functional requirement is that the semantics of the plan model should be readily understandable to trained humans. The syntax does not need to be human-friendly since user-friendly interfaces can be built to generate syntax from user actions that convey the semantics. Since computers can handle a wide variety of syntax, however, it should be possible to design a syntax that is friendly to both humans and computers. That will be helpful when no user-friendly interface is available and a human needs to do planning.

6. CONCLUSION

We plan to build:

- an OWL model of kitting workstation process plans
- example process plans conforming to the model
- C++ software for representing, reading, writing, and accessing the plans
- a C++ kitting workstation plan executor
- a simulated kitting workstation
- an actual kitting workstation

Using the simulated and actual workstations, we plan to evaluate the performance of the kitting process plan model. Where we find a need for additional functionality in the model, it will be added. If we discover plan functionality that is not used in our example plans and does not appear likely to be used in any plans, it will be removed.

We intend to include sensory processing in the kitting workstation. Some of this, such as a switch that detects whether a gripper is seated properly in a gripper changer, might be used only by the robot controller. Other sensory data will be reported to the workstation's knowledge base. For example, we might have fixed cameras that feed into a system that computes the observed locations of objects in the workstation. For any observed object, the observed location data might be fused with the location data that is a priori or entered in the course of plan execution. A large difference between the stored and observed values might trigger an error signal.

The sensory processing described in the previous paragraph requires nothing from the contents of a process plan or from a process planner. The only thing it requires from a process plan executor is the ability to receive error signals and react to them. Other elements of the system would handle sensory processing and knowledge base maintenance. Hence, we currently do not deal with sensory processing in the process plan model.

If we find that the robot needs to help with sensory processing or that sensory devices need explicit instructions that are coordinated with robot actions, then the process plan model will need to be expanded to include tasks for sensory processing devices or robot tasks that serve sensory processing. For example, a camera end effector might be defined and used. If a part were dropped and could not be found by fixed sensors, the robot would change to

the camera end effector and move it into position to see where fixed cameras cannot see. As another example, if a box with one part in it is dropped and the part cannot be located, the robot might be commanded to move the box in order to determine if the part is now under the box.

As mentioned earlier, there are currently no kitting workstation tasks in the process plan model designed specifically for error recovery. If it is found that error recovery tasks are needed in the process plan model, they will be added.

7. REFERENCES

[1] ISO 1998. *Industrial automation systems and integration – Product data representation and exchange – Part 49: Integrated generic resources: Process structure and properties*, 1998, International Organization for Standardization.

[2] ISO 2004. *Industrial automation systems and integration – Physical device control – Data model for computerized numerical controllers -- Part 10: General process data*, 2004, International Organization for Standardization.

[3] ISO 2005. *Industrial automation systems and integration – Product data representation and exchange – Part 240: Application protocol: Process plans for machined products*, 2005, International Organization for Standardization.

[4] ISO 2007. *Industrial automation systems and integration – Product data representation and exchange – Part 238: Application protocol: Application interpreted model for computerized numerical controllers*, 2007, International Organization for Standardization.

[5] W3C 2009. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax 4.1*, W3C Editor's Draft 21 September 2009, W3C, http://www.w3.org/2007/OWL/draft/ED-owl2-syntax-20090921.

[6] Dimensional Measurement Standards Consortium 2009. *Dimensional Measuring Interface Standard Part 1 Revision 5.2*, ANSI/DMIS 105.2 Part 1-2009, Dimensional Measurement Standards Consortium.

[7] SRI 2008. *OWL-S 1.2 Release*, http://www.ai.sri.com/daml/services/owl-s/1.2.

[8] Nof, S. Y., Wilhelm, W. E., Warnecke, H-J., 1997. *Industrial Assembly* , Chapman & Hall, London, UK.

[9] Ghallab, M., Nau, D., Traverso, P., 2004. *Automated Planning Theory and Practice*, Morgan Kaufmann, San Francisco, USA.

[10] Catron, B., Ray, S., 1991. *ALPS – A Language for Process Specification*, International Journal of Computer Integrated Manufacturing, Vol. 4, No. 2, pp 105-113.

[11] http://sourceforge.net/apps/mediawiki/owl-cpp

[12] Wallace, S., Senehi, M. K., Barkmeyer, E., Ray, S., Wallace, E., 1993. *Manufacturing Systems Integration Control Entity Interface Specification*, NISTIR 5272, National Institute of Standards and Technology, Gaithersburg, MD, USA.

[13] Kramer, T. R., Horst, J. A., Huang, H. M., Messina, E., Proctor, F. M., Scott, H. A., 2004. *Feature-Based Inspection and Control System*, NISTIR 7098, National Institute of Standards and Technology, Gaithersburg, MD, USA.

[14] Jasthi, S. R. K., Rao, P. N., Tewari, N. K., 1995. *Studies on Process Plan Representation in CAPP systems*, Computer Integrated Manufacturing Systems, Vol. 8, No. 3, pp 173-184.

[15] Steele, G. L, 1990. *Common LISP the Language*, Second Edition, Digital Equipment Corporation.

[16] ISO/IEC 14882:2011. *Information Technology – Programming Languages – C++*, 2011, International Organization for Standardization.

[17] Kramer, T. R., Proctor, F., Xu X., Michaloski, J. L., 2006. *Run-time interpretation of STEP-NC: implementation and performance*; International Journal of Computer Integrated Manufacturing, Volume 19, Issue 6, pp 495 – 507.

[18] W3C 2004. XML Schema Part 1: Structures Second Edition, W3C Recommendation 28 October 2004, http://www.w3.org/TR/xmlschema-1.