Knowledge Driven Robotics

Stephen Balakirsky[b], Zeid Kootbally[b,c], Thomas Kramer[a,b], Anthony Pietromartire[b], Craig Schlenoff[b]

[a]*Catholic University of America, Washington, DC, USA*
[b]*National Institute of Standards and Technology, Gaithersburg, MD USA*
[c]*University of Maryland, College Park, MD, USA*

Abstract

In this article, a newly developed knowledge architecture that was designed to support the IEEE Robotics and Automation Society's Ontologies for Robotics and Automation Working Group will be presented. This architecture allows for the creation of systems that demonstrate flexibility, agility, and the ability to be rapidly re-tasked. The architecture, as well as an implementation that combines extensions to the Unified System for Automation and Robot Simulation (USARSim) and the Robot Operating System (ROS), will be discussed in detail. This implementation extends USARSim and ROS to provide models of industrial robots and end-effectors while providing a case study in the area of robotic kit building. The architecture's potential for creating flexible, agile, and rapidly re-taskable systems will be presented, and possible areas of future work will be discussed.

Keywords: Don't forget to add keywords

1. Introduction

Today's state-of-the-art industrial robots are often programmed with a teach pendent. This requires that the robot cell be taken off-line for a human-led teaching period when the cell's task is altered. Many research systems depend on numerous hand-tuned parameters that must be re-tuned by operators whenever a untested environment is encountered. These requirements for human intervention cause the robotic systems to be brittle and limited in operational scope. The robotic systems of tomorrow need to be capable, flexible, and agile. These systems need to perform their duties at least as well as human counterparts, be able to be quickly re-tasked to other operations, and be able to cope with a wide variety of unexpected environmental and operational changes. In order to be successful, these systems need to combine domain expertise, knowledge of their own skills and limitations, and both semantic and geometric environmental information.

The IEEE Robotics and Automation Society's Ontologies for Robotics and Automation Working group is striving to create an overall ontology and associated methodology for knowledge representation and reasoning that will address these knowledge needs. As part of the Industrial Subgroup of the IEEE Working Group, the authors have been examining a novel architecture that allows for the combination of the static aspects of the ontology with dynamic sensor processing to allow for a robotic system that is able to cope with environmental and task changes without operator intervention.

Architectures have always been an important part of advanced robotic systems and have evolved with the robotic systems that they characterize. Early architectures such as the one developed for the Army Research Laboratory's (ARL) Demo I program tended to be hardware focused [1]. The Demo I platform was able to perform reconnaissance, surveillance, and target acquisition (RSTA) as well as tele-operated control over a compressed bandwidth radio channel. However, the lack of a software architecture limited the sharing of information between software applications and forced the human operator to fuse information returned from various applications in order to make control decisions.

The ARL Demo III program demonstrated sophisticated off-road navigation while incorporating a much more sophisticated hardware/software architecture known as 4D/RCS [2]. This architecture provide for a closer coupling of software systems and allowed for the sharing of items such as sensor data and cost maps. However, the lack of a knowledge architecture limited this system's ability to utilize *a priori* knowledge to reason over its environment and provide fine-grained classification of terrain and obstacles. This resulted in numerous parameters that required human tuning based on the expected environmental conditions.

In order to become accepted in industrial applications, tomorrow's systems will need to do more. They will have to demonstrate flexibility through the elimination of hand-tuned parameters. They will need to demonstrate agility through the ability to cope with rapidly changing situations and task requirements, and they will need to be able to be rapidly re-tasked in order to adapt to new tasking quickly and efficiently. The industrial subgroup of the Ontologies for Robotics and Automation Working group is striving to demonstrate that the inclusion of knowledge in the form of an ontology will allow robotic systems to progress to this next level.

The organization of the remainder of this paper is as follows: Section 2 presents an overview of the knowledge driven methodology that has been developed for this effort. Section 3 describes the domain of kit building which is a greatly simplified, but still practically useful manufacturing/assembly domain. Section 4 describes our ontology for the kitting domain. Section 5 describes the implementation of the system and Section 6 provides for conclusions and future work.

2. Design Methodology

The design approach described in this article is not intended to replace sound engineering of an intelligent system, but rather as an additional step that may be applied in order to provide the system with more agility, flexibility, and the ability to be rapidly re-tasked. This is accomplished by assuring that the appropriate knowledge of the correct scope and format is available to all modules of the intelligent system.

The overall knowledge model of the system may be seen in Figure 1. The figure is organized vertically by the representation that is used for the knowledge and horizontally by the classical sense-model-act paradigm of intelligent systems. On the vertical axis, knowledge begins with Domain Specific Information that captures operational knowledge that is necessary to be successful in the particular domain in which the system is designed to operate. This information is then organized into a domain independent representation (an Ontology) that allows for the encoding of an object taxonomy, object-to-object relationships, and aspects of actions, preconditions, and effects. Aspects of this knowledge are automatically extracted and encoded in a form that is optimized for a planning system to utilize (the Planning Language). Once a plan has been formulated, the knowledge is transformed into a representation that is optimized for use by a robotic system (the Robot Language).

It is acknowledged that sensing and action are important parts of a robotic system. However, this article focuses on knowledge representation, and thus the modeling section will be described in the most detail.

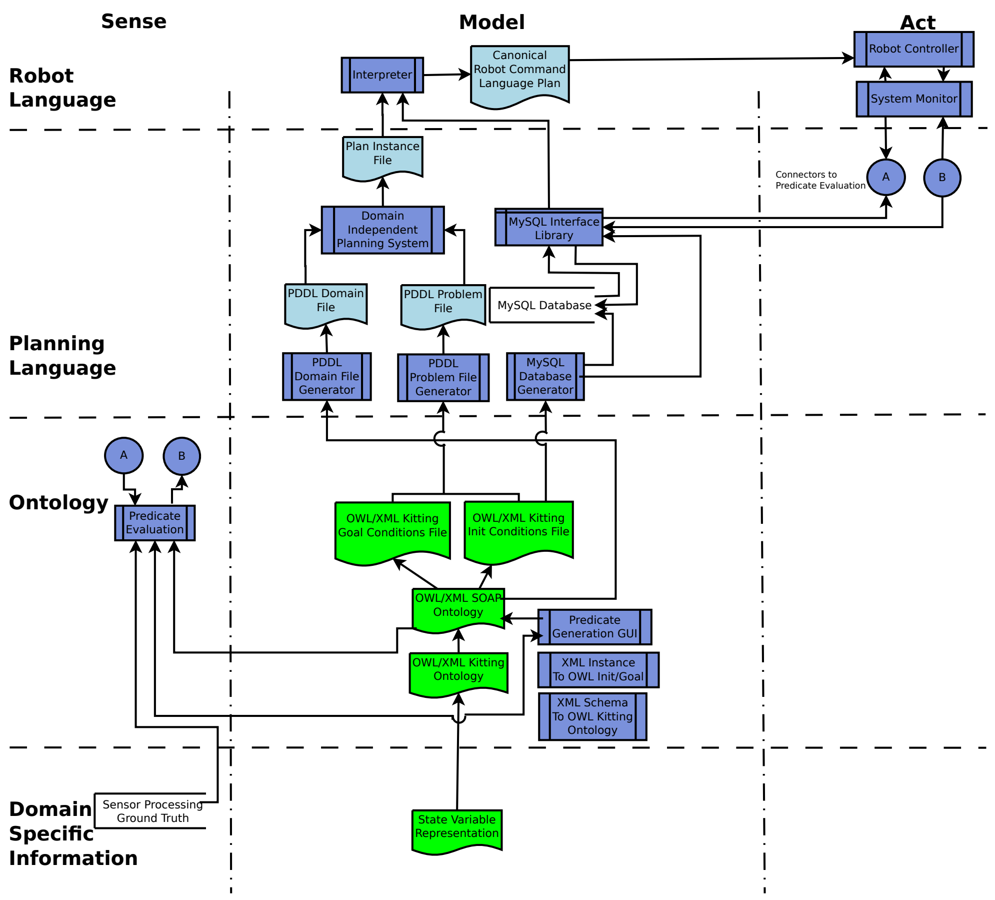
Sense
Model
Act
Robot Language
Planning Language
Ontology
Domain Specific Information
Interpreter
Canonical Robot Command Language Plan
Robot Controller
System Monitor
Plan Instance File
Connectors to Predicate Evaluation
A
B
Domain Independent Planning System
MySQL Interface Library
PDDL Domain File
PDDL Problem File
MySQL Database
PDDL Domain File Generator
PDDL Problem File Generator
MySQL Database Generator
A
B
Predicate Evaluation
OWL/XML Kitting Goal Conditions File
OWL/XML Kitting Init Conditions File
OWL/XML SOAP Ontology
Predicate Generation GUI
OWL/XML Kitting Ontology
XML Instance To OWL Init/Goal
XML Schema To OWL Kitting Ontology
Sensor Processing Ground Truth
State Variable Representation

Figure 1: Knowledge Driven Design extensions – In this figure, green shaded boxes with curved bottoms represent hand generated files while light blue shaded boxes with curved bottoms represent automatically created boxes. Square boxes represent processes and libraries.

2.1. Domain Specific Information

The most basic knowledge that must be gathered for a knowledge driven system is domain specific information (DSI). This appears along the bottom row of Figure 1. DSI includes sensors and sensor processing that are specifically tuned to operate in the target domain. Examples of sensor processing may include pose determination and object identification.

For the knowledge model, a scenario driven approach is taken where the DSI design begins with a domain expert creating one or more use cases and specific scenarios that describe the typical operation of the system. Based on these scenarios and use cases, the high-level actions that the system must be able to accomplish can be enumerated and described. An action description that includes any preconditions that must be true for an action to be valid as well as expected effects that will result from a given action is then created for each action.

put-kittray(*robot*,*kittray*,*worktable*): The *Robot robot* puts the *KitTray kittray* on the *WorkTable worktable*.

preconditions	*effects*
kittray-location-robot(*kittray*,*robot*)	¬kittray-location-robot(*kittray*,*robot*)
robot-holds-kittray(*robot*,*kittray*)	¬robot-holds-kittray(*robot*,*kittray*)
worktable-empty(*worktable*)	¬worktable-empty(*worktable*)
	kittray-location-worktable(*kittray*,*worktable*)
	robot-empty(*robot*)
	on-worktable-kittray(*worktable*,*kittray*)

Figure 2: Example action along with its preconditions and expected effects.

Based on the action description, objects in the environment that are relevant for system operation can be identified. For example, the action depicted in Figure 2 has a given robot place a kit tray onto a work table. The preconditions for this action are that the robot is holding a kit tray and there is a clear space on the table to place the tray. This is represented by the predicate expressions shown in Figure 2 that specify that the robot is holding the kit tray, the kit tray is located on the robot (actually in its gripper), and the work table is empty. The expected effects of this action are that the kit tray is now located on the work table and the robot is no longer holding it. This is also represented by a series of predicates that are shown in Figure 2. Based on the preconditions and expected effects, the objects that are relevant to this action include the robot, the kit tray, and the work table. Aspects of these items may now be represented in the DSI. For example, that a kit tray has a location and may be held by the robot or placed on the work table.

2.2. Ontology

The design transitions from domain expertise to knowledge modeling expertise when the State Variable Representation (SVR) is used to generate an ontology which consists of three parts:

1. A base ontology that describes the objects in the scenario. This file contains all of the basic information that was determined to be needed during the evaluation of the use cases and scenarios. The knowledge is represented in as compact of a form as possible with knowledge classes inheriting common attributes from parent classes. For example, the class for a work table, WorkTable is derived from BoxyObject, and BoxyObject is derived from SolidObject. The actual size of the work table is defined in the BoxyObject. The WorkTable includes all of the attributes of a BoxyObject along with the notion that a work table is something that can have other objects placed on it. The work table itself is part of a work station WorkStation.
2. Extensions that describe the States of the world and relationships between states, Ordering constructs, Actions, and Predicates (the *SOAP* ontology) that are relevant to the scenario. This extension contains not only the basic class for actions and predicates, but also the individual actions and predicates that are necessary for the domain under study. In the case of the kit building domain, it was found that 10 actions and 16 predicates were necessary.
3. Instance files that describe the initial and goal states for the system. The final set of files for the ontology contain a description of the complete system starting or initial state and the requirements on the goal state. The initial state file must contain a description of the environment that is complete enough for a planning system to be able to create a valid sequence of actions that will achieve the given goal state. For the kit building domain, this includes information such as the location of the various end effectors that are available to the robot, the locations and contents of part supply bins, and the location where finished kits should be placed. The goal state file only needs to contain information that is relevant to the end goal of the system. For the case of building a kit, this may simply be that a complete kit is located in a bin designed to hold completed kits.

The ontology files are described in more detail in Section 4.

2.3. Planning Language

The Planning Domain Definition Language (PDDL) [3] is an attempt by the domain independent planning community to formulate a standard language for planning. A community of planning researchers has been producing planning systems that comply with this formalism since the first International Planning Competition held in 1998. This competition series continues today, with the seventh competition being held in 2011. PDDL is constantly adding extensions to the base language in order to represent more expressive problem domains. The work represented in this article is based on PDDL Version 3. By placing the knowledge in a PDDL representation, the use of an entire family of open source planning systems is enabled.

The PDDL input format consists of two files that specify the domain and the problem. As shown in Figure 1, these files are automatically generated from the ontology. The domain file represents actions along with required preconditions and expected results. The problem file represents the initial state of the system and the desired goal.

From these two files, a domain independent planning system such as the forward-chaining partial-order planning system from Coles *et. al* [4] may be run to create a static plan file. This plan file contains a sequence of actions that will transition the system from the initial state to the goal state. In order to maintain flexibility, it is desired that detailed information that is subject to change should be "late-bound" to the plan. In other words, specific information is acquired directly before that information needs to be used by the system. This allows for last minute changes in this information. For example, the location of a kit tray on a work table may be different from run to run. However, one would like to be able to use the same planning sequence for constructing the kit independent of the tray's exact position. To compensate for this lack of exact knowledge, the plans that are generated by the PDDL planning system contain only high-level actions.

```
1 (attach-endeffector robot_1 tray_gripper tray_gripper_holder changing_station_1)
2 (take-kittray robot_1 kit_tray_1 empty_kit_tray_supply tray_gripper work_table_1)
3 (put-kittray robot_1 kit_tray_1 work_table_1)
4 ...
5 (put-part robot_1 part_a_1 kit_a2b2c1 work_table_1 part_a_tray)
6 (remove-endeffector robot_1 part_gripper part_gripper_holder changing_station_1)
7 (attach-endeffector robot_1 tray_gripper tray_gripper_holder changing_station_1)
8 (take-kit robot_1 kit_a2b2c1 work_table_1 tray_gripper)
9 (put-kit robot_1 kit_a2b2c1 finished_kit_receiver)
```

Figure 3: Excerpt of the PDDL solution file for kitting.

Figure 3 depicts an excerpt of a solution for the construction of a kit. Line 3 with the command *put-kittray* shows the command that is issued to place a kit tray onto the work table. To facilitate late binding, this command does not specify the exact location of the work table. This kind of knowledge detail is maintained by sensor processing and is stored in a MySQL database [5]. As shown in Figure 1, the actual tables for the database are auto-generated during the design phase and the knowledge is utilized in combination with the PDDL planned action by the interpreter in order to form Robot Language Commands.

2.4. Robot Language

While the high-level PDDL commands are a convenient representation for the planning system, they do not contain the information that is required by a robotic controller to successfully control a robotic cell. The interpreter combines knowledge from the PDDL plan file with knowledge from the MySQL database to form a sequence of sequential actions that the robot controller is able to execute. The authors devised a canonical robot command language (CRCL) in which such lists can be written. The purpose of the CRCL is to provide generic commands that implement the functionality of typical industrial robots without being specific either to the language of the planning system that makes a plan or to the language used by a robot controller that executes a plan.

It was anticipated that plans being generated by this system would be executed by a variety of robot controllers using robot-specific languages for input programs. The authors themselves are using a ROS controller [6] to control a robot[1]. The controller and the interpreter 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.

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. More information on CRCL commands may be found in Balakirsky *et. al* [7].

[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.

3. Industrial Kitting

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 [8] "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 [9].

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 [10] including saving manufacturing or assembly space [11], reducing assembly workers walking and searching times [12], and increasing line flexibility [8] and balance [13].

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.

4. Ontology

We believe that building models of the world knowledge is a necessary step towards operating an automated kitting workstation. The proposed models include representations for non-executable information about the kitting workstation such as information about parts, kits, and trays. The description of these models includes for instance the location, orientation, and relation between components. These models are discussed in Section 4.1. Models of executable information are also produced from the system information described in the SVR. These models include actions along with their precondition and effect, and failures. Section 4.2 discusses models of executable information.

Models for non-executable and executable information can be combined to generate the OWL/XML kitting *init* and *goal* conditions files, as described in Section 4.3.

4.1. The OWL/XML Kitting Ontology

In order to maintain compatibility with the IEEE Robotics and Automation Society's Ontologies for Robotics and Automation Working Group, the *Kitting* ontology has been fully defined in the Web Ontology Language (OWL) [14]. In addition, the ontology was also fully defined in the XML schema language [15]. 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) that are discussed in Balakirsky *et al.* [16].

Table 1: Excerpt of the *Kitting* ontology.

SolidObject *PrimaryLocation SecondaryLocation*
Kit *Tray DesignRef Parts Finished?*
LargeBoxWithEmptyKitTrays *LargeContainer Trays*
LargeBoxWithKits *LargeContainer Kits KitDesignRef Capacity*
PartsTrayWithParts *PartTray*
...
DataThing
PhysicalLocation *RefObject*
PoseLocation *Point XAxis ZAxis*
PoseLocationIn
PoseLocationOn
RelativeLocation *Description*
RelativeLocationIn
RelativeLocationOn
...

SolidObject and DataThing constitute the two top-level classes of the *Kitting* ontology model, from which all other classes are derived. SolidObject models solid objects, things made of matter. The *Kitting* ontology includes several subclasses of SolidObject that are formed from components that are SolidObject. The DataThing class models data for SolidObject. Examples of subclasses for SolidObject and DataThing are represented in Table 1. Items in italics following classes are names of class attributes. Derived types inherit the attributes of the parent. Each attribute has a specific type not shown in Table 1. If an attribute type has derived types, any of the derived types may be used.

Using Table 1, an example of interaction between classes SolidObject and DataThing can be expressed as follows: Each SolidObject A has at least one PhysicalLocation (the *PrimaryLocation*). A PhysicalLocation is defined by giving a reference SolidObject B (*RefObject*) and information saying how the position of A is related to B. PhysicalLocation consists of two types of location which are required for the operation of the kitting workstation:

- Mathematically precise locations are needed to support robot motion. The mathematical location, PoseLocation, gives the pose of the coordinate system of A in the coordinate system of B. The mathematical information consists of the location of the origin of A's coordinate system (*Point*) and the directions of its Z (*ZAxis*) and X (*XAxis*) axes. The mathematical location variety has subclasses representing that, in addition, A is in B (PoseLocationIn) or on B (PoseLocationOn).
- Relative locations (class RelativeLocation), specifically the knowledge that one SolidObject is in (RelativeLocationIn) or on (RelativeLocationOn) another, are needed to support making logical plans for building kits. The subclasses of RelativeLocation are needed not only for logical planning, but also for cases when the relative location is known, but the mathematical information is not available.

4.2. The OWL/XML SOAP Ontology

As depicted in Figure 1, the *SOAP* ontology imports the *Kitting* ontology. The *Kitting* ontology is involved in the process that generates the PDDL domain file and in the process that evaluates the truth-value of predicates. While some concepts in the *SOAP* ontology are used by both processes, other concepts are exclusive to one of these two processes. We approach the description of the *SOAP* ontology with a discussion on each process.

4.2.1. PDDL Domain File Generator Process

A PDDL domain file consists of definitions of predicates, functions, and actions. Actions are ways of changing the state of the world and consist of a precondition and an effect sections which consist of predicates and functions. Predicates are used to encode Boolean state variables while functions are used to model updates of numerical values. Introducing functions into planning makes it possible to model actions in a more compact and sometimes more natural way [17]. Both predicates and functions are well documented in the SVR. Figure 4 shows the structure of a domain file.

```
 1 (define (domain kitting-domain)
 2    (:requirements :strips :typing :derived-predicates :action-costs :fluents)
 3    (:types
 4       EndEffector
 5       EndEffectorHolder
 6       ...
 7    )
 8 (:predicates
 9    (endeffector-location-robot ?endeffector - EndEffector ?robot - Robot)
10    ...
11 )
12 (:functions
13    (quantity-partstray ?partstray - PartsTray)
14    ...
15 )
16 (:action put-part
17    :parameters(
18       ?robot - Robot
19       ?part - Part
20       ?kit - Kit
21       ?worktable - WorkTable
22       ?partstray - PartsTray)
23    :precondition (and
24       (part-location-robot ?part ?robot)
25       (robot-holds-part ?robot ?part)
26       (on-worktable-kit ?worktable ?kit)
27       (origin-part ?part ?partstray)
28       (< (quantity-kit ?kit ?partstray)
29       (capacity-kit ?kit ?partstray))
30       (kit-location-worktable ?kit ?worktable))
31    :effect (and
32       (not (part-location-robot ?part ?robot))
33       (not (robot-holds-part ?robot ?part))
34       (part-not-searched)
35       (not (found-part ?part ?partstray))
36       (part-location-kit ?part ?kit)
37       (increase (quantity-kit ?kit ?partstray) 1)
38       (robot-empty ?robot))
39 )
40 ...
41 )
```

Figure 4: PDDL action put-part.

The PDDL domain file consists of a unique name (`kitting-domain` at line 1). The `requirements` section (line 2) defines the features that are used in the domain file. The `types` section (lines 3–7) shows the different types of objects that are used in predicates, functions, and actions. The `predicates` section (lines 8–11) and the `functions` section (lines 12–15) list all the predicates and functions, respectively, that are used to build an action. Both predicates and functions have a unique name, parameters (defined by the "?" sign) and types of parameters (placed after the "-" sign). The `action` section (lines 16–39) formulates the core of an action. The different components of an `action` are described as follows: The unique name of the action (`put-part` in this example) comes directly after the keyword `:action`. The `parameters` section (lines 17–22) list all the parameters that participate in this action. The `precondition` section (lines 23–30) lists the predicates that must be true and the operations on the functions. The `effect` section (lines 31–38) lists the predicates that must be true and the operations on the functions.

The representation of a PDDL domain file in the ontology requires OWL classes definitions of the components described in Figure 4 and the definition of the relations between these components. The different classes required for the representation of a PDDL domain file are Domain, Predicate, Function, Action, Precondition, Effect, FunctionBool, and FunctionOperation. These classes are subclasses of the class DataThing. The relations between these classes are described below.

1. Domain – A Domain has a name (*hasDomain_Name*) of type `string`, at least one requirement (*hasDomain_Requirement*) of type `string`, and at least a type (*hasDomain_Type*) of type `string`. A Domain can consist of only one Predicate (*hasDomain_Predicate*), only one Function (*hasDomain_Function*), or a combination of both. A Domain has at least one Action (*hasDomain_Action*).
2. Action – An Action has a Precondition (*hasAction_Precondition*) and an Effect (*hasAction_Effect*). An Action has a ParameterList (*hasAction_ParameterList*) that contains all the parameters for a PDDL action. As seen in Figure 4, an action has unique name (*hasAction_Name*) of type `string`.
3. ParameterList – The *put-part* action illustrated in Figure 4 has five parameters of different types. Each one of these types is represented by a class in the *Kitting* ontology. To represent all PDDL actions in the *SOAP* ontology, we considered all the different types of parameters that are used in all our ten PDDL actions. To date, we are using eleven different types of parameter, represented by the eleven following classes: Robot, EndEffectorChangingStation, KitTray, Kit, LargeBoxWith-

EmptyKitTrays, LargeBoxWithKits, WorkTable, PartsTray, Part, EndEffector, and EndEffectorHolder. Therefore, ParameterList has at least a parameter (*hasAction_Parameter*) that is from one of these eleven classes.

The order of the parameters in a PDDL action also needs to be represented in the ontology. In Figure 4, the parameter `robot` comes before the parameter `part`, the parameter `part` comes before the parameter `kit`, and so on. OWL has no built-in structure to represent an ordered list, instead, all the eleven classes mentioned earlier, use *hasParameter_Next* to point to the next parameter in ParameterList.

4. Precondition – A Precondition can consist of Predicate (*hasPrecondition_Predicate*) and Function (*hasPrecondition_Function*). An Effect can contain o or more FunctionBool (*hasPrecondition_FunctionBool*). A Precondition belongs to one Action (*hadByPrecondition_Action*).
5. Effect – An Effect can consist of Predicate (*hasEffect_Predicate*) and Function (*hasEffect_Function*). An Effect can contain o or more FunctionBool (*hasEffect_FunctionBool*). An Effect can contain 0 or more FunctionOperation. An Effect belongs to one Action (*hadByEffect_Action*). A negative Predicate is represented with the OWL built-in property assertion `owl:NegativePropertyAssertion`.
6. Predicate – A Predicate has a unique name (*hasPredicate_Name*) of type `string`. A Predicate has a reference parameter (*hasPredicate_RefParam*) and a target parameter (*hasPredicate_TargetParam*). A reference parameter is the first parameter in the Predicate's list and the target parameter is the second parameter in the parameter's list. A Predicate cannot have more than two parameters due to the definition of Predicates in the SVR. In the case a Predicate has only one parameter, it is assign to the reference parameter. Reference and target parameters are one of the parameters defined for the Action to which the Predicate belongs.
7. Function – A Function has a unique name (*hasFunction_Name*) of type `string`. A Function has a reference parameter (*hasFunction_RefParam*) and a target parameter (*hasFunction_TargetParam*). The same rules apply to the definition and use of these two types of parameters as the ones defined for Predicate.
8. FunctionBool – A FunctionBool is used to represent functions comparison such as the one depicted at lines 28–29 in Figure 4. A FunctionBool has one or more subclasses that represent the type of relation between

two Functions. For example, the relation depicted at line 13–14 in Figure 4 is represented in the subclass IntLesserThanInt. Subclasses of FunctionBool have a first Function (*hasFunctionBool_FirstFunction*) that represents the Function on the left side of the operator. Subclasses of FunctionBool have a second Function (*hasFunctionBool_SecondFunction*) that represents the Function on the right side of the operator.

9. FunctionOperation – A FunctionOperation represents a function operation such as the one depicted at line 37 in Figure 4. A FunctionOperation consists of subclasses for each type of operation and FunctionOperation has one Function (*hasFunctionOperation_Function*).

4.2.2. The Predicate Evaluation Process

The Predicate Evaluation process is used to identify action failures during plan execution by the robot. The identification of action failures relies on spatial relations to compute the truth value of each predicate in each action involved in the plan. Action failures and spatial relations are discussed in the remainder of this section.

Representation of Action Failure. A failure is any change or any design or manufacturing error that renders a component, assembly, or system incapable of performing its intended function. In kitting, failures can occur for multiple reasons: equipment not set up properly, tools and/or fixtures not properly prepared, lack of safety, and improper equipment maintenance. Part/component availability failures can be triggered by inaccurate information on the location of the part, part damage, wrong type of part, or part shortage due to delays in internal logistics. In order to prevent or minimize failures, a disciplined approach needs to be implemented to identify the different ways a process design can fail before impacting the productivity.

Failures detected in the workstation can result in the current plan to become obsolete. When a failure is detected in the execution process and the failure mode identified, the value of the severity for the failure mode will be retrieved from the ontology and the appropriate contingency plan will be activated. 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. To select the right contingency plan, i.e., the less time consuming or safer, the system will need to rely on the information from the knowledge representation.

In the knowledge driven diagram (Figure 1), the Predicate Evaluation process is responsible for failure detection. An action failure consists of failure modes that can occur during the execution of a PDDL action. The steps to identify action failures in the kitting system are described in Figure 5.

```
1  for each action $\mathcal{A}$ in the Plan Instance File do
2      Interpreter;
3      converts $\mathcal{A}$ into a set $\mathcal{S}$ of Canonical Robot Language commands;
4      stores $\mathcal{S}$ in Canonical Robot Language Plan;
5  end
6  for each set $\mathcal{S}$ in Canonical Robot Language Plan do
7      Robot Controller reads $\mathcal{S}$;
8      System Monitor calls Predicate Evaluation;
9      Predicate Evaluation;
10      traces back action $\mathcal{A}$ from set $\mathcal{S}$;
11      computes truth-value of predicates for action $\mathcal{A}$ precondition;
12     if output of Predicate Evaluation is true then
13         Robot Controller executes $\mathcal{S}$;
14         Predicate Evaluation computes truth-value of predicates for
           action $\mathcal{A}$ effect;
15         if output of Predicate Evaluation is true then
16             end;
17         end
18         else failure;
19     end
20     else failure;
21 end
```

Figure 5: Failure identification.

As seen in Figure 5, failures are identified during the execution of canonical robot commands (line 6), generated from PDDL actions (line 3) by the Interpreter. The Predicate Evaluation process outputs a Boolean value that results in a failure detection if this value is false (lines 18 and 20). Therefore, to represent failures in the *SOAP* ontology, the following concepts are introduced:

- "Action" the PDDL action for which a failure can occur.
- "Failure Modes": List of failure modes that can occur during the execution of a specific action.
- "Causes" of failure: Causes can be of different types, such as components, usage conditions, human interaction, internal factors, external factors, etc.

- "Predicates" that can be responsible for the occurrence of the "Failure Mode".
- "Effects" of the failure: Consequences associated to the failure mode.
- "Severity" of the "Effect(s)": Assessment of how serious the effects would be should the failure occur. Each effect is given a rank of severity ranging from 1 (minor) to 10 (very high). The severity rank is used to trigger the appropriate contingency plan.
- "Probability of Occurrence": an estimate number of frequencies (based on experience) that a failure will occur for a specific action.

Table 2 shows an example of failure modes associated to the PDDL action *put-part*(*robot*,*part*,*kit*,*worktable*,*partstray*) which is defined as "The *Robot robot* puts the *Part part* in the *Kit kit*".

Table 2: Failure modes for the PDDL action *put-part*.

Action	Failure Mode(s)	Cause(s)	Effect(s)	Severity	Occurrence (%)	Predicate(s)
put-part	part falls off of the end effector	end effector hardware issues	downtime	9	60	part-location-robot robot-holds-part
			part damage	5		
	part not released at all	end effector hardware issues	downtime	9	8	¬(part-location-robot) ¬(robot-holds-part) robot-empty
		wrong/inexistant canonical command	downtime	7		

The column *Predicate(s)* shows the predicates from the action *put-part* that can activate the corresponding failure mode (column *Failure Mode(s)*) if their truth-value is evaluated to false.

The classes discussed below are used to represent action failures in the *SOAP* ontology. All these classes are subclasses of DataThing.

1. Action – An Action has at least one FailureMode (*hasAction_FailureMode*).
2. FailureMode – A FailureMode has at least one FailureEffect (*hasFailureMode_FailureEffect*). A FailureMode has a description (*hasFailureMode_Description*) of type `Literal` which represents the nature of the failure mode. The cause of the failure mode is expressed with *hasFailureMode_Cause* and is of type `Literal`. The occurrence of the failure mode is expressed with *hasFailureMode_Occurrence* and is of

type `Integer`. A FailureMode has at least one Predicate (*hasFailureMode_Predicate*) that can be responsible for the occurrence of the failure mode. The Predicate should be already defined *prior* to its association with the class FailureMode.

3. FailureEffect – A FailureEffect has one failure severity (*hasFailureEffect_FailureSeverity*) of type `integer` and a description (*hasFailureEffect_Description*) of type `Literal`.

The next section discusses how state relations are used to identify action failures.

Spatial Relation Representation. As seen in the action failure section, the Predicate Evaluation process is called by the System Monitor process to check the truth-value of a predicate. The output of this process is a Boolean value that is redirected back to the System Monitor. We have represented the concept of "Spatial Relations" in the *SOAP* ontology so that any algorithm can compute the truth-value of a predicate.

"Spatial Relations" are represented as subclasses of the RelativeLocation class which is a subtype of the PhysicalLocation (see Table 1). There are three types of spatial relations, each represented in a separate class as described below:

- RCC8_Relation: RCC8 [18] is a well-known and cited approach for representing the relationship between two regions in Euclidean space or in a topological space. Based on the definition of RCC8, the class RCC8_Relation consists of eight possible relations, including Tangential Proper Part (TPP), Non-Tangential Proper Part(NTPP), Disconnected (DC), Tangential Proper Part Inverse (TPPi), Non-Tangential Proper Part Inverse (NTPPi), Externally Connected (EC), Equal (EQ), and Partially Overlapping (PO). In order to represent these relations in all three dimensions for the kitting domain, we have extended RCC8 to a three-dimensions space by applying it along all three planes (x-y, x-z, y-z) and by including cardinal direction relations "+" and "-" [19]. In the ontology, RCC8 relations and cardinal direction relations are represented as subclasses of the class RCC8_Relation. Examples of such classes are X-DC, X-EC, X-Minus, and X-Plus.

- Intermediate_State_Relation: These are intermediate level state relations that can be inferred from the combination of RCC8 and cardinal direction relations. For instance, the intermediate state relation

Contained-In is used to describe object *obj1* completely inside object *obj2* and is represented with the following combination of RCC8 relations:

$$\begin{gathered}\textbf{Contained-In}(\mathit{obj1}, \mathit{obj2}) \rightarrow \\ (\texttt{x-TPP}(\mathit{obj1}, \mathit{obj2}) \vee \texttt{x-NTPP}(\mathit{obj1}, \mathit{obj2})) \wedge \\ (\texttt{y-TPP}(\mathit{obj1}, \mathit{obj2}) \vee \texttt{y-NTPP}(\mathit{obj1}, \mathit{obj2})) \wedge \\ (\texttt{z-TPP}(\mathit{obj1}, \mathit{obj2}) \vee \texttt{z-NTPP}(\mathit{obj1}, \mathit{obj2}))\end{gathered}$$

In the ontology, intermediate state relations are represented with the OWL built-in property `owl:equivalentClass` that links the description of the class Intermediate_State_Relation to a logical expression based on RCC8 relations from the class RCC8_Relation.

- Predicate: The representation of predicates has been illustrated in Section 4.2.1. In this section we discuss how the class Predicate has been extended to include the concept of "Spatial Relation". The truth-value of predicates can be determined through the logical combination of intermediate state relations. The predicate kit-location-lbwk(*kit*, *lbwk*) is true if and only if the location of the kit *kit* is in the large box with kits *lbwk*. This predicate can be described using the following combination of intermediate state relations:

  $$\begin{gathered}\textsf{kit-location-lbwk}(\mathit{kit}, \mathit{lbwk}) \rightarrow \\ \textbf{In-Contact-With}(\mathit{kit}, \mathit{lbwk}) \wedge \\ \textbf{Contained-In}(\mathit{kit}, \mathit{lbwk})\end{gathered}$$

  As with state relations, the truth-value of predicates is captured in the ontology using the `owl:equivalentClass` property that links the description of the class Predicate to the logical combination of intermediate state relations from the class Intermediate_State_Relation.

  As seen in Section 4.2.1, a predicate can have a maximum of two parameters. In the case where a predicate has two parameters, the parameters are passed to the intermediate state relations defined for the predicate, and are in turn passed to the RCC8 relations were the truth-value of these relations are computed. In the case the predicate has only one parameter, the truth-value of intermediate state relations, and by inference, the truth-value of RCC8 relations will be tested with this parameter and with every object in the environment in lieu of the second parameter. Our kitting domain consists of only one predicate that has no parameters. This predicate is used as a flag in order to force some actions to come before others during the formulation of the plan. Predicates of this nature are not treated in the concept of "Spatial Relation".

4.3. The OWL/XML Kitting Init and Goal Conditions File

The OWL/XML kitting *init* and *goal* conditions files are used to build the PDDL problem file. This section first describes how a PDDL problem file is structured and then reviews the classes used in the different ontologies to build the PDDL problem file.

4.3.1. Init and Goal sections in the Problem File

Figure 6 is a fragment of the problem file generated for our kitting system where we have emphasized the init and goal sections.

```
 1 (define (problem kitting-problem)
 2    (:domain kitting-domain)
 3    (:objects
 4       robot_1 - Robot
 5       kit_tray_1 - KitTray
 6       kit_a2b2c1 - Kit
 7       ...
 8         )
 9     (:init
10       (robot-with-no-endeffector robot_1)
11       (partstray-not-empty part_a_tray)
12       ...
13       (= (capacity-kit kit_a2b2c1 part_a_tray) 2)
14       (= (capacity-kit kit_a2b2c1 part_b_tray) 2)
15       ...)
16     (:goal (and
17     (= (quantity-kit kit_a2b2c1 part_a_tray) (capacity-kit kit_a2b2c1 part_a_tray))
18     ...
19     (kit-location-lbwk kit_a2b2c1 finished_kit_receiver))
20 )
```

Figure 6: Excerpt of a PDDL problem file for kitting.

The different sections that form the problem file are described as follows. The keyword `problem` (line 1) signals a planner that the current file contains all the elements required to constitute a PDDL problem file. The keyword `domain` (line 2) refers to the domain file that the current problem file depends on. The `objects` section (lines 3–7) declare all the objects present in the world. Some of these objects are required in both the initial state and the goal state. The `:init` keyword (line 9) signals a planner about the initial state of the world. The predicates that are true and the initial values for functions (lines 10–15) are listed in the `init` section. The `:goal` keyword (line 16) signals a planner about the final state of the world. The predicates defined in this section must be true and the functions must have the defined numerical values in the final state.

4.3.2. Representation of the Init and Goal Files Using Ontologies

In our kitting system, the `init` and `goal` sections always consist of predicates and functions. The following steps describe how we build the OWL/XML kitting *init* and *goal* conditions files.

1. In the *SOAP* ontology, instances of the objects that will appear in the `:objects` section of the problem file are created. Since these objects are used in both the `init` and `goal` sections in the generated PDDL problem file, it is important to make these objects available to the OWL/XML kitting *init* and *goal* conditions files. For instance, the object `robot_1` of type `Robot` (line 4 in Figure 6) will be generated from the instance `robot_1` in the class Robot, created in the *SOAP* ontology.
2. Predicates and functions that will appear in the `init` section for the generated PDDL problem file are created in the OWL/XML kitting *init* file as instances of the class Predicate and Function, respectively.
3. Predicates and functions that will appear in the `goal` section in the generated PDDL problem file are created in the OWL/XML kitting *goal* file as instances of the class Predicate and Function, respectively.
4. Instances of the class Predicate and of the class Function created in the OWL/XML kitting *init* and *goal* conditions files point to parameters (instances of classes) that have been created in the *SOAP* ontology (step 1).

5. Implementation

6. Conclusion

Future work: Represent contingency plans in the ontology.

References

[1] S. Balakirsky, P. David, P. Emmerman, F. Fisher, P. Gaylord, in: Proc. Symposium of the Association for Unmanned Vehicle Systems, pp. 927–946.

[2] J. Albus, in: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pp. 3260–3265.

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

[4] A. J. Coles, A. Coles, M. Fox, D. Long, in: 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, AAAI 2010, Toronto, Ontario, Canada, pp. 42–49.

[5] O. Corporation, in: www.mysql.com.

[6] W. Garage, in: http://www.willowgarage.com/pages/software/ros-platform.

[7] S. Balakirsky, T. Kramer, Z. Kootbally, A. Pietromartire, Metrics and Test Methods for Industrial Kit Building, Technical Report, National Institute of Standards and Technology (NIST), 2012.

[8] Y. A. Bozer, L. F. McGinnis, International Journal of Production Economics 28 (1992) 1–19.

[9] A. Schyja, A. Hypki, B. Kuhlenkotter, in: Robotics and Automation (ICRA), 2012 IEEE International Conference on, pp. 5246 –5251.

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

[11] L. Medbo, International Journal of Production Economics Journal of Industrial Ergonomics 31 (2003) 263–281.

[12] G. Schwind, Material Handling Engineering 47 (1992) 43–45.

[13] J. Jiao, M. M. Tseng, Q. Ma, Y. Zou, Concurrent Engineering: Research and Applications 8 (2000) 297–321.

[14] F. Harmelen, D. McGuiness, in: http://www.w3.org/TR/2004/REC-owl-features-20040210/.

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

[16] S. Balakirsky, T. Kramer, Z. Kootbally, A. Pietromartire, C. Schlenoff, The Industrial Kitting Ontology Version 0.5, White Paper, National Institute of Standards and Technology, Gaithersburg, MD, 2012.

[17] M. Fox, D. Long, Journal of Artificial Intelligence Research (JAIR) 20 (2003) 61–124.

[18] F. Wolter, M. Zakharyaschev, in: Proceedings of the 7th Conference on Principles of Knowledge Representation and Reasoning, KR2000, Morgan Kaufmann, 2000, pp. 3–14.

[19] C. Schlenoff, A. Pietromartire, Z. Kootbally, S. Balakirsky, T. Kramer, S. Foufou, Engineering Creative Design in Robotics and Mechatronics.