An Ontology Based Approach to Action Verification for Agile Manufacturing

Stephen Balakirsky[1] and Zeid Kootbally[2]

[1] Georgia Tech Research Institute, Atlanta, GA 30332, USA,
stephen.balakirsky@gtri.gatech.edu,
WWW home page: unmannedsystems.gtri.gatech.edu
[2] University of Maryland, College Park, MD 20740, USA,
zeid.kootbally@umd.edu,
WWW home page: www.nist.gov/el/isd/ks/kootbally.cfm

Abstract. Many of today's robotic work cells are unable to detect when an action failure has occurred. This results in faulty products being sent down the line, and/or downtime for the cell as failures are detected and corrected. This article examines a novel knowledge-driven system that provides added agility by detecting and correcting action failures. The system also provides for late binding of action parameters, thus providing flexibility by allowing plans to adapt to changing environmental conditions. The key feature of this system is its knowledge base that contains the necessary relationships and representations to allow for failure detection and correction. This article presents the ontology that stores this knowledge as well as the overall system architecture. The manufacturing domain of kit construction is examined as a sample test environment.

Keywords: failure detection, manufacturing, ontology, robotics, Planning Domain Definition Language

1 Introduction

A failure is any change, design, or manufacturing error that renders a component, assembly, or system incapable of performing its intended function [6]. In kitting, as described in Section 2, failures can occur for multiple reasons that include equipment not being set up properly, tools and/or fixtures not being properly prepared, and improper equipment maintenance. Part/component availability failures can be triggered by inaccurate information on the location of the part, part damage, incorrect part types, 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 and to allow for corrective actions to be taken before the failure impacts productivity.

Even though today's state-of-the-art industrial robots are capable of sub-millimeter accuracy [7], they often lack the sensing necessary to detect failures and the programming required to cope with and correct the failure. This is due to the fact that they are often programmed by an operator using imprecise

positional controls from a teach pendant. These teach pendant programs are highly repeatable, which provides utility for large-batch, error-free operation. However, the cyclic program that repeats identical operations does not lend itself well to adaptation for failure mitigation. In fact, producing a program to correct a perceived failure would require that the cell be taken off-line for additional human-led teach pendant programming. In addition, most cells lack the ability to sense that a failure occurred and lack programming (that would have had to be teach pendant entered) to cope with failure conditions, thus making it impossible for the cell to recover from failures. This leads to faulty products being sent down the line, and/or downtime for the cell as failures are detected and corrected.

For small batch processors or other customers who must frequently change their line configuration or desire to perform complex operations with their robots, this frequent downtime and lack of failure correction/detection may be unacceptable. 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 quickly re-tasked to other operations, cope with a wide variety of unexpected environmental and operational changes, and be able to detect and correct errors in operation. 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 has taken the first steps in creating the infrastructure necessary for such a system, while the Industrial Subgroup has applied this infrastructure to create a sample kit building system. This work is presented in Balakirsky et al. [2] which describes the construction of a robotic kit building system that is able to cope with environmental and task changes without operator intervention. This article extends that work to utilize the same infrastructure to allow for the detection and correction of action failures in the system.

The organization of the remainder of this paper is as follows. Section 2 describes the domain of kit building. Section 3 presents an overview of the software system architecture as well as details of the ontology and world model for the robot cell. Section 4 discusses the detailed operation of cell, and Section 5 discusses how failures are handled by the ontology. Finally, Section 6 presents conclusions and future work.

2 Kitting

Today's advanced manufacturing plants utilize mixed-model assembly where multiple product variants are built on the same line. According to Jim Tetreault, Ford's vice president of North America Manufacturing, new Ford[3] assembly fa-

[3] No approval or endorsement of any commercial product by the authors is intended or implied. Certain commercial software systems are identified in this paper to facilitate understanding. Such identification does not imply that these software systems are necessarily the best available for the purpose.

cilities are able to build a full spectrum of vehicles on the same assembly line [9]. One of the technologies that makes this possible is the use of assembly kits. Bozer and McGinnis [4] describe a kit as "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". These kits provide a synchronous material flow, where parts and components move to assembly stations in a just-in-time manner. The kits provide workers with the parts and tools that they need (which may vary from vehicle model to vehicle model) in the sequence that they need them. The use of kitting also allows a single delivery system to feed multiple assembly stations thus saving manufacturing or assembly space [10] and provides an additional inspection opportunity that allows for the detection of part defects before they impact assembly operations. The individual operations of the station that builds the kits may be viewed as a specialization of the general bin-picking problem [11] where parts are picked from one or more part bins or trays and placed into specific slots in a kit tray.

For our sample implementation, we assume that the robot cell is building one of several possible kit configurations. At execution time, the cell has a set kit to build, but does not know the precise location of the kit tray, the part trays, or the location of individual parts in the part tray. When a human builds a kit, they are able to inspect each part before adding it to the kit tray. This provides an additional level of quality control and is an aspect that is desirable to have in our robotic system. During kit construction, 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, inspect them, and place them in the kit.
3. Pick up the completed kit and place it in the full kit storage area.

Each of these steps may be a compound action that includes other actions such as end-of-arm tool changes, path planning, and obstacle avoidance. The items that are being placed in the kit may be of varying size and shape and have various grasping and inspection requirements.

3 System Overview

The kitting system that has been implemented as part of this work is a deliberative intelligent system based on the 4D/RCS reference model architecture [1]. This architecture is a hierarchical architecture in which each echelon or level follows a sense-model-act paradigm. The basic structure of the system may be seen in Figure 1.

3.1 Sense

In order to sense action failures associated with kit building, it is necessary to be able to detect the six-degree of freedom pose of relevant objects in the world. One issue with pose detection is the large number of potential target

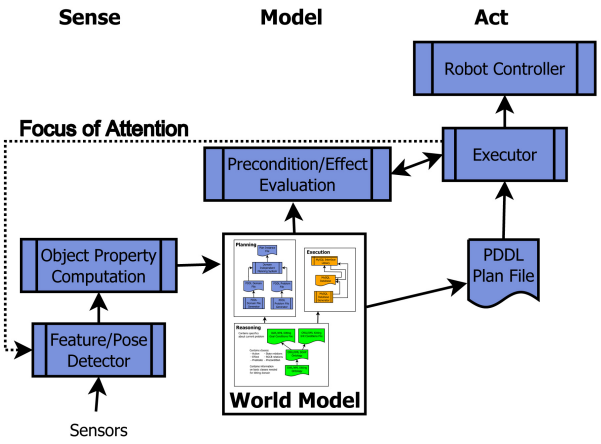

Fig. 1. Major components that make up the Sense–Model–Act paradigm of the kitting station.

objects and object classes in the world. Both the number of objects and potential classes can be reduced by intelligently selecting critical objects of interest that are tagged with predicted locations and object classes for the sensor system to track. This object selection, also known as focus of attention, is guided by the Executor process with knowledge obtained from preconditions and effects of planned actions. Actual algorithm development for pose and object detection is an active research area, and is beyond the scope of this article. For our purposes, we have assumed the use of a high-quality system that is capable of recognizing a limited variety of items in a controlled environment and then determining each item's pose.

3.2 Model

The world model that is being utilized is shown in Figure 2. The model contains knowledge that is structured specifically for reasoning, planning, and execution. All of the concepts necessary for the manufacturing domain under test are encoded in the ontology that resides in the reasoning section of the model. The planning and execution sections of the model are automatically generated from this section.

Ontology – The reasoning portion of the world model is designed to contain all of the information needed to reason over and solve complex manufacturing problems. The knowledge is represented in a Web Ontology Language (OWL) ontology that is structured in three parts. The first part of the ontology contains generic information and classes that are needed for the domain of kit building. This area of the ontology contains information on basic elements such as a

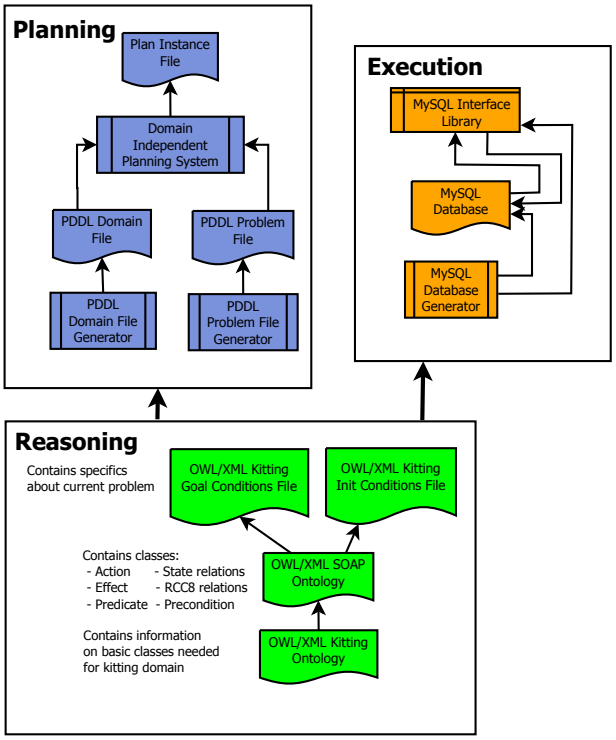

Fig. 2. System World Model - The world model contains a Reasoning section that is based on an ontology shown in green, a Planning section that is based on a Planning Domain Definition Language (PDDL) specification shown in blue, and an Execution section that is based on a relational database (MySQL) shown in orange.

"point" which is defined as a class that contains a name and a three-dimensional quantity, as well as complex types such as a "part", which is shown in Figure 3, and contains elements such as the part's location and a name that references a stock keeping unit. The stock keeping unit contains static information on classes of parts such as the part's shape, weight, and the end effector that should be used for grasping the part. This information is utilized to create parameters for the Planning World Model and the skeleton tables for the MySQL database of the Execution World Model.

Both static and dynamic information is represented in this ontology and is automatically transitioned into the Planning and Execution areas of the world model. During system operation, dynamic information is updated in the Execution World Model. More information on this portion of the ontology may be found in [3].

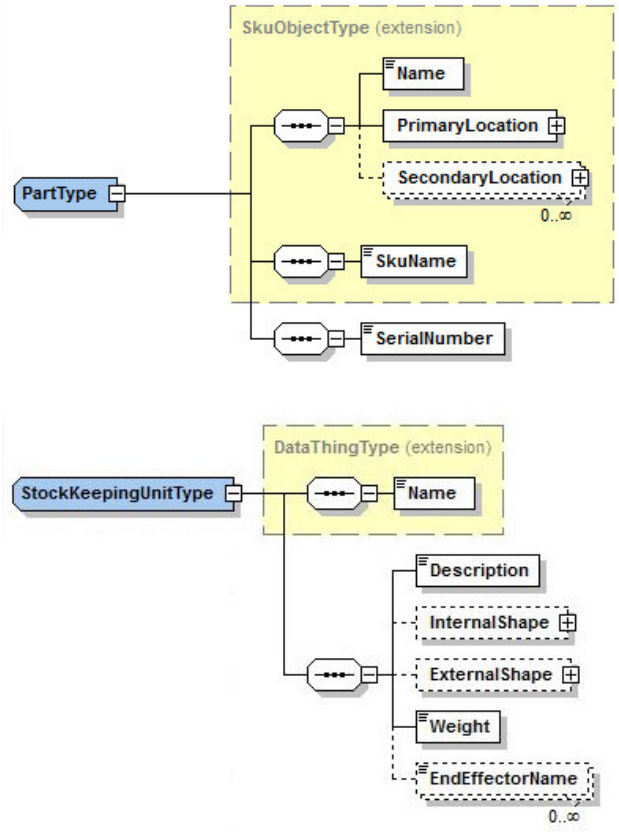

Fig. 3. Description of the PartType class that is designed to contain both static and dynamic information about particular parts and the StockKeepingUnitType class that contains static information about classes of parts.

The second part of the ontology (known as States, Ordering constraints, Actions, and Predicates or SOAP) contains the high-level concept of an action and all of the concepts that are required to support an action. In this case, a PDDL [8] action is being represented, and this action is defined as an operator that causes one or more properties of an instance to change. Before this action may be performed, certain preconditions must be satisfied, and after the action is performed, certain effects will take place. The action accepts parameters that specify the particular instances that will be affected, where an instance refers to a physical object or piece of grounded data in the world. All of the necessary information for the automatic generation of the PDDL domain file required by the planning system is contained in this section of the ontology. The classes used to represent the actions in the ontology are provided in the enumerated list shown below.

1. *Action* – An *Action* is the basic type that is used by a planning system to produce changes on the world. It has an *ActionPrecondition* that must be valid before the action may be executed and an *ActionEffect* that is expected to be produced by the action. Each action requires parameters that are used to specify which objects are being operated upon. These parameters are contained in the *ActionParameterList*. An *Action* has a unique name.
2. *ActionParameterList* – Actions may have multiple parameters of different types that are represented by a class in the upper ontology. The *ActionParameterList* contains the particular action's parameters that are instances from the upper ontology. The order of the parameters in an action also needs to be represented in the ontology. OWL has no built-in structure to represent an ordered list. In order to maintain parameter ordering, the parameter uses *hasNextParameter* and *hasPreviousParameter* to point to the next and the previous parameter in *ParameterList*, respectively.
3. *ActionPrecondition* – An *ActionPrecondition* specifies necessary conditions that must be true in order for an action to be undertaken. It can consist of an *ActionPredicate*, an *ActionFunction*, an *ActionFunctionBool*, or a combination of these three classes. An *ActionPrecondition* belongs to one *Action*.
4. *ActionEffect* – An *ActionEffect* specifies the results that are anticipated to occur as a result of a particular action. It can consist of an *ActionPredicate*, an *ActionFunction*, an *ActionFunctionBool*, or a combination of these three classes. An *ActionEffect* belongs to one *Action*. A negative *ActionPredicate* is represented with the declaration of *hasEffect_Predicate* within the OWL built-in property assertion `owl:NegativePropertyAssertion`.
5. *ActionPredicate* – A predicate is used to specify a binary property of a single object, or a relationship between two objects. For example, the predicate `(robot-empty ?robot)` is true if the robot `?robot` is not holding anything. The predicate `(part-location-robot ?part ?robot)` is true only if the reference parameter `?part` is being held by the target parameter `?robot`. An *ActionPredicate* represents these predicates. It has a unique name of type `string`, a reference parameter and an optional target parameter. The reference parameter is the first parameter in the predicate's parameter list and the target parameter is the second parameter in the predicate's parameter list. An *ActionPredicate* cannot have more than two parameters due to the inherent definition of predicates. In the case where an *ActionPredicate* has only one parameter, it is assigned to the reference parameter.
6. *ActionFunction* – In version 3 of the PDDL language, it is possible to use numeric functions that contain one or two parameters. For example, the function `(quantity-partstray ?partstray - PartsTray)` will return the number of parts that are contained in the parts tray `?partstray`, and the function `(quantity-kit ?kit - Kit ?partstray - PartsTray)` will return the number of parts from parts tray `?partstray` that are currently in the kit `?kit`. The class *ActionFunction* is used to represent these functions. It has a unique name of type `string` along with a reference parameter and a target parameter. The same rules apply to the definition and use of these two types of parameters as defined for *ActionPredicate*.

7. *ActionFunctionBool* – In version 3 of the PDDL language, it is also possible to compare the results returned by two functions. The class *ActionFunctionBool* is used to represent these relationships. It has one or more subclasses that represent the type of relation (mathematical operator) between two *ActionFunction*s. Subclasses of *ActionFunctionBool* have a first *ActionFunction* that represents the *ActionFunction* on the left side of the operator and a second *ActionFunction* that represents the *ActionFunction* on the right side of the operator.

The third part of the ontology contains specific instances needed for a particular kitting domain. For example, it will contain the definition of the finished kits that may be constructed and specific information on the individual parts. One of the goals of this framework is to introduce additional agility into the kit building process. Therefore, partial information is accepted and even encouraged for this area of the ontology. For the example of a part shown in Figure 3, information on the SKU, grasp points (part of the ExternalShape or InternalShape), and name would be expected to be available at runtime. Information on the location of the part (PrimaryLocation) may not become valid until after a sensor processing system has identified and located the particular part.

Planning – PDDL 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 representation in the world model 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 such as the forward-chaining partial-order planning system from Coles et al. [5] is enabled. In order to operate, the PDDL planners require a PDDL file-set that consists of two files that specify the domain and the problem. From these files, the planning system creates an additional static plan file. Both the domain and problem file are able to be auto-generated from the ontology.

The generated static 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.

As seen in Figure 2, the planning world model framework contains generators that read the ontology and create a standard PDDL domain and PDDL problem files. Any of the family of PDDL Version 3 compatible planning systems is then able to be run on these files to create the static plan instance file. A representation of this plan may be stored in the ontology for future use.

Execution – The execution world model is also built automatically from the ontology. This world model consists of a MySQL database and C++ interfaces that provide for easy access to the data. The table skeletons are generated from the kitting ontology, and the tables are initially populated with information from the initial condition file. During plan execution, the Executor guides the sensor processing system in updating the information in this section of the world model. All of the data structures encoded in the ontology are included in this representation.

3.3 Act

The actions that take place in the kitting work cell are coordinated by the Executor as illustrated in Figure 1. The Executor reads PDDL actions as input and outputs a standardized set of low-level robot commands encoded in a language developed at the National Institute of Standards and Technology (NIST) known as the Canonical Robotic Command Language (CRCL) [3].

Before and after each high-level command is executed, the Executor sends focus of attention information into the sensor processing system. This allows the sensor processing system to compute the appropriate predicate relations that are required to verify the conditions necessary to carry out an action and that an action's execution has been successful. Information on predicates is written to the world model by the sensing system and read from the world model by the Executor.

4 System Operation

In order to construct a kit, the kitting system steps through each action in the precomputed PDDL plan. Failures are searched for both before and after execution of each action. The overall process, known as BuildKit is shown in Figure 4.

This process begins by retrieving a planning instance that has been precomputed to solve the construction of the requested kit (Line 1 of Figure 4). This is a high-level PDDL plan that is not grounded to actual part instances or locations. It contains information on the named storage location for classes of parts (individual SKU numbers), the quantity of each SKU that is required by the kit, and a build order (a sequence of SKUs to be added to the kit). Additional information on the appropriate end-of-arm tooling required to grip each part is also included.

```
   Data: kitToBuild
   Result: reports success or failure
 1 retrieve instance PDDLInstance to construct kit kitToBuild;
 2 for each action A in PDDLInstance do
 3     for each precondition P of action A do
 4         if PredicateEvaluation(P) = false then
 5             report failure;
 6         end
 7     end
 8     create set S of Canonical Robot Control Language Commands;
 9     send set S to Robot Controller for execution;
10     for each effect E of action A do
11         if PredicateEvaluation(E) == false then
12             report failure;
13         end
14     end
15     report action success;
16 end
17 report plan success;
```

Fig. 4. BuildKit – Sequences the actions necessary to build a kit.

This planning instance is decomposed into individual actions that must be successfully carried out to complete the construction of the kit (the *for* loop beginning at Line 2 of Figure 4). At this point, preconditions of the action are examined to assure that the action to be attempted is valid. If any of the action's preconditions are not able to be validated, a failure is reported; otherwise, the action is approved for execution.

```
   Data: PredicateIn
   Result: true or false
 1 determine predicted pose PoseR of PredicateIn.ReferenceParameter and
   PoseT of PredicateIn.TargetParameter ;
 2 send PredicateIn, pose PoseR, and PoseT as focus of attention command to
   SensorProcessing;
 3 if Eval(PredicateIn) == true then
 4     return true;
 5 else
 6     return false ;
 7 end
```

Fig. 5. PredicateEvaluation – Returns the truth value of the predicate expression.

4.1 Precondition Validation

Each precondition is a predicate expression that must be validated prior to action execution. The procedure for validating predicates is shown in Figure 5. In Line 1 of this algorithm, the world model is queried for the pose and class of each relevant parameter of the predicate. The information returned is the latest knowledge that has been recorded by the sensor processing system and is not guaranteed to be up-to-date. This possibly out-of-date information is used as a prediction of the object's current pose and the knowledge is sent as a focus of attention indicator to the sensor processing system. The sensor processing system is instructed to update the world model with current observations and to compute the supporting relationships necessary for predicate evaluation. Figure

Data: $PredicateIn, PoseR, PoseT$
1 determine actual pose $APoseR$ of $PredicateIn.ReferenceParameter$;
2 determine actual pose $APoseT$ of $PredicateIn.TargetParameter$;
3 determine RCC8 relations between $APoseR$ and $APoseT$;
4 determine Intermediate State Relations based on RCC8 relations;
5 determine truth value of $PredicateIn$ and write to MySQL database;

Fig. 6. SensorProcessing – Updates the MySQL database in the Execution world model to contain the latest evaluation of predicates related to $PredicateIn$.

6 depicts the algorithm that is followed by sensor processing in the computation of predicate values. As may be seen from this figure, the computation of predicates is a three step process that involves the computation of increasingly complex forms of spatial relations. These relationships; Region Connection Calculus (RCC8) relations, intermediate state relations, and predicate relations, are each represented as a separate class in the ontology.

RCC8 Relation – RCC8 [12] is an approach for representing the relationship between two regions in Euclidean or topological space. The class *RCC8_Relation* is based on the definition of RCC8 and consists of eight possible relations that include 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, RCC8 has been extended to a three-dimensional space by applying it along all three planes (x-y, x-z, y-z) and by including cardinal direction relations "+" and "-". In the ontology, RCC8 relations and cardinal direction relations are represented as subclasses of the class *RCC8_Relation*.

Intermediate State Relation – These relations can be inferred from the combination of RCC8 and cardinal direction relations. For instance, the inter-

mediate state relation **In-Contact-With** is used to describe that object *obj1* is in contact with object *obj2*. This is true if *obj1* is externally connected to *obj2* in the x-direction, the y-direction, or the z-direction, and is represented with the following combination of RCC8 relations:

$$\textbf{In-Contact-With}(obj1, obj2) \rightarrow$$
$$\texttt{x-EC}(obj1, obj2) \vee \texttt{y-EC}(obj1, obj2) \vee \texttt{z-EC}(obj1, obj2)$$

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 Relation – The truth-value of predicates can be determined through the logical combination of intermediate state relations. The predicate *endeffector-location-endeffectorholder*(*endeffector*, *endeffectorholder*) is true if and only if the location of the end effector *endeffector* is in the end effector holder *endeffectorholder* and is not attached to a robot. This predicate can be described using the following combination of intermediate state relations:

$$\text{endeffector-location-endeffectorholder}(endeffector, endeffectorholder) \rightarrow$$
$$\textbf{In-Contact-With}(endeffector, enfectorholder) \wedge$$
$$\neg\textbf{In-Contact-With}(endeffector, robot)$$

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*.

Truth Value Determination – As seen in Section 3.2, 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 where the truth-value of these relations are computed. If 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".

These truth values may be retrieved from the ontology for use in Line 3 of Figure 5 which will then propagate back up to BuildKit. If the predicates are successfully evaluated, the action will be cleared for execution and a set of CRCL Commands will be generated.

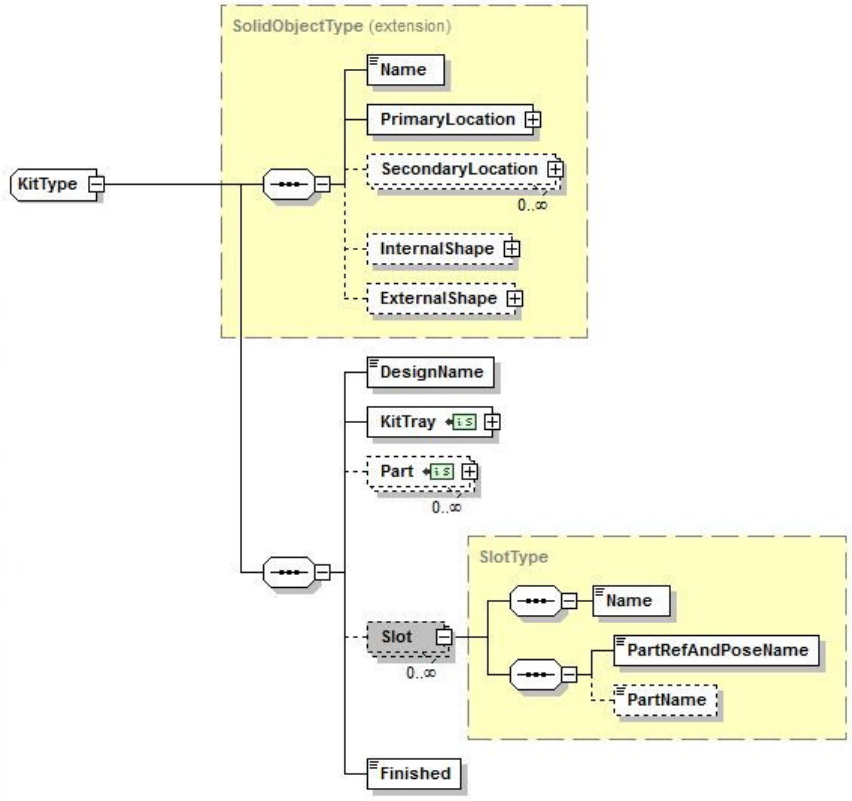

Fig. 7. Description of the KitType class that is designed to bring together a container (the KitTray), a design for the kit creation, and specific parts that populate the kit.

4.2 Canonical Robot Command Language Generation

Up to this point, the PDDL actions are not fully grounded to specific instances that exist in the world. For example, the action `put-part` is designed to place the part that is currently being held by the robot into a kit that is specified as one of the action's input parameters. However, the precise location for this part to be placed is not specified at the time of plan creation. It is up to the Executor, working with the world model, to find an empty slot in the kit that can receive the part. The structure of the ontology is specifically designed to support this grounding, and this structure is automatically replicated in the MySQL database that resides in the Execution portion of the world model.

Continuing with the example of the `put-part` command, the Executor needs to find an empty slot in the target kit to place the specific part that the robot is holding. As shown in Figure 7, the KitType class contains zero to many Slot classes that in turn contain specific location and part identification information. The Executor is able to read this information from the world model and determine the precise global pose where the part should be placed. The robot controller must now be commanded to complete this action.

While the action is an atomic element in PDDL, it will decompose into a series of actions in CRCL. The robot will need to follow a safe trajectory to achieve the slot in the kit, and the gripper will need to be controlled in order to release the part. This one-to-many mapping is performed in the Executor and is currently hand-coded for each of the PDDL commands that exists in the system.

4.3 Effect Validation

The purpose of performing an action is to achieve results in the world. These results are represented in the PDDL effects. Each effect is a predicate expression that must be validated to assure proper action execution. The technique for validating the effect predicates is identical to the evaluation of the precondition predicates described in Section 4.1. If all of the effects are able to be validated, the system will report the action's success and begin performing the next action in the plan.

5 Failure Analysis

As seen in Figure 5, failures are identified during the evaluation of preconditions and effects. In addition to recognizing failures, it is desired to be able to respond to them. In order to properly model this response, additional information must be modeled in the ontology. As shown in Figure 8, the class *ActionFailureType* has been added to the ontology. This class is composed of a *PredicateType* class and a *FailureModeType* class. It contains all of the information required to be able to diagnose and remediate any failures that are detected by the system.

The *PredicateType* class contains a list of predicates whose truth values caused the detection of the current failure condition. Examination of the instances pointed to by the *ReferenceParameter* and the *TargetParameter* allow the system to understand exactly which components were involved in the failure.

The *FailureModeType* class provides various known failure modes that could exist for the combination of *PredicateType*s that were found deficient. It provides the consequences of such a failure occurring, information on how to remediate such a failure, and the chance that this kind of failure could occur. Understanding the consequences of the failure mode is important if one would like to be able to pinpoint the correct cause of the failure. For example, assume that the action TAKE-PART failed, and the predicate that indicated the failure was *part-location-robot*.

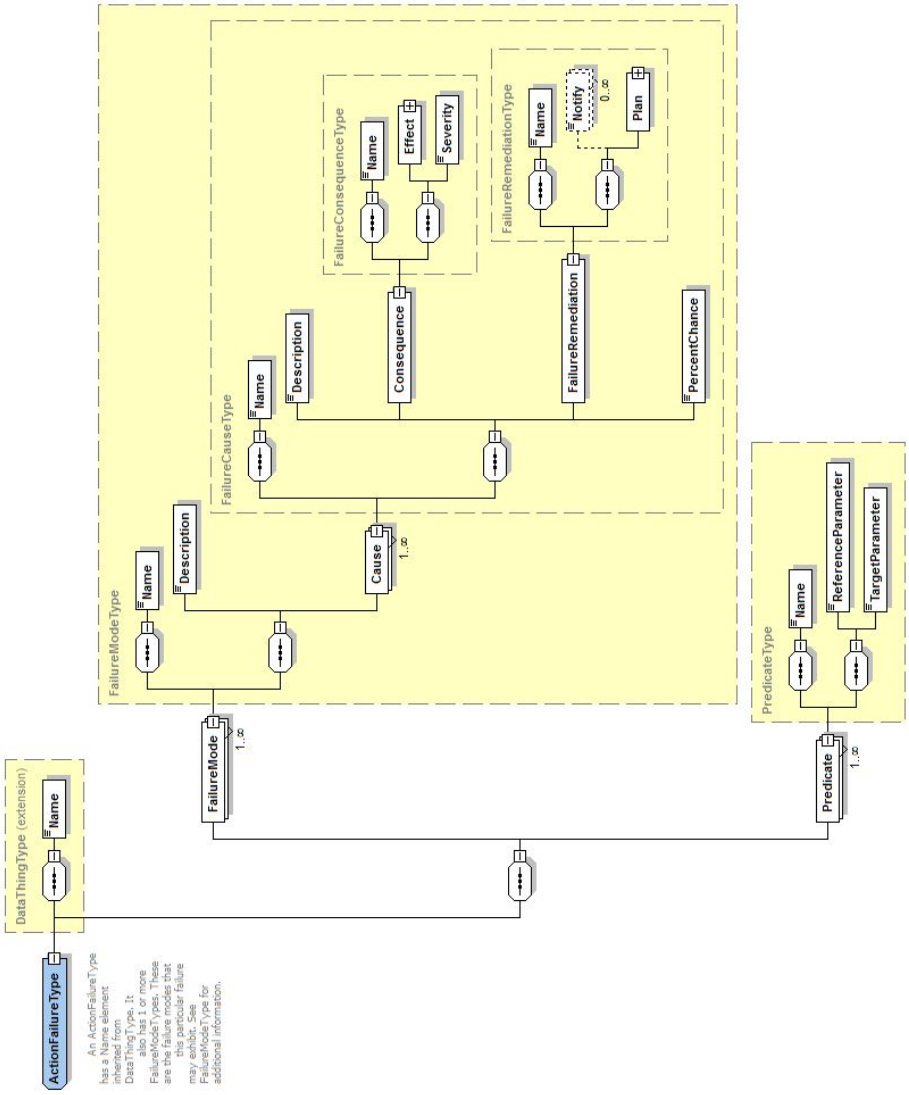

Fig. 8. Description of the FailureType Class that is used to store information on action failures.

Understanding where the part is actually located is critical to understanding the root cause of the failure. If the part is still in the parts tray, it would indicate a grasp related failure. If the part is on the floor of the cell, it would indicate a part handling error. This kind of information is represented in the *Effect* class of the consequences. Being able to pinpoint the effects of the failure also allows for judgment on the failure severity. A missed grasp will likely lead to a new grasp

attempt and will have little impact on operation. A dropped part may cause damage to the part and have a higher severity level.

Failure remediation is described by the *FailureRemediation* class. This class contains information on what notifications need to be sent as a result of the failure as well as what actions should be taken. Continuing our example from above, the grasp failure may cause grasping information to be sent to the enginering department for algorithm refinement while the dropped part may notify logistics that an additional component will be required. Recovery from the failure is possible by executing the *Plan* that accompanies the detected failure.

6 Conclusions and Future Work

The framework described in this paper has been applied to the domain of kit building, which is a simple, but practically useful manufacturing/assembly domain. Through its use, we have been able to demonstrate agility in both kit construction through late binding of part locations, and in recovery from action failures through the detection of failures and ability to compensate for the failure's effects.

There are several areas in the system that still utilize hand-coding of data. It is desired that extensions to the ontology be created that will allow for the automatic application of knowledge and eliminate code that is specifically tuned to a particular set of predicates or actions. The hand-coded areas include the conversion of PDDL actions to CRCL sequences as well as the retrieval of instance data from the MySQL database for predicate evaluation. Work is currently underway to correct for these deficiencies.

Extensions are also possible that will expand this work to the realm of general assembly. We hope to apply this knowledge based framework to simple assembly tasks (growing towards more complex tasks) on a real robot workcell in the near future.

References

1. J. Albus. 4-d/rcs reference model architecture for unmanned ground vehicles. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3260–3265, 2000.
2. S. Balakirsky, Z. Kootbally, T. Kramer, A. Pietromartire, C. Schlenoff, and S. Gupta. Knowledge driven robotics for kitting applications. *Robotics and Autonomous Systems*, pages –, 2013.
3. S. Balakirsky, T. Kramer, Z. Kootbally, and A. Pietromartire. Metrics and test methods for industrial kit building. NISTIR 7942, National Institute of Standards and Technology (NIST), 2012.
4. Y. A. Bozer and L. F. McGinnis. Kitting versus line stocking: A conceptual framework and descriptive model. *International Journal of Production Economics*, 28:1–19, 1992.

5. A. J. Coles, A. Coles, M. Fox, and D. Long. Forward-chaining partial-order planning. In *20th International Conference on Automated Planning and Scheduling, ICAPS 2010*, pages 42–49, Toronto, Ontario, Canada, May 12–16 2010. AAAI 2010.
6. J. Collins. *Failure of Materials in Mechanical Design: Analysis, Prediction, Prevention*. Wiley Interscience, 1993.
7. Control and Montreal Robotics Lab at the ETS. Measuring the absolute accuracy of an abb irb 1600 industrial robot. *http://www.youtube.com/watch?v=d3fCkS5xFlg*, 2011.
8. M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl–the planning domain definition language. Technical Report CVC TR98-003/DCS TR-1165, Yale, 1998.
9. T. James. Ford's michigan eco car plant: one size fits all. *Engineering and Technology Magazine*, 7, July 2011.
10. L. Medbo. Assembly work execution and materials kit functionality in parallel flow assembly systems. *International Journal of Industrial Ergonomics*, 31:263–281, 2003.
11. A. Schyja, A. Hypki, and B. Kuhlenkotter. A modular and extensible framework for real and virtual bin-picking environments. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 5246 –5251, may 2012.
12. F. Wolter and M. Zakharyaschev. Spatio-temporal representation and reasoning based on rcc-8. In *Proceedings of the 7th Conference on Principles of Knowledge Representation and Reasoning*, KR2000, pages 3–14. Morgan Kaufmann, 2000.