NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

Intelligent Systems Division

Action Failures Identification

December 9, 2012

Stephen Balakirsky	stephen.balakirsky@nist.gov
Zeid Kootbally	zeid.kootbally@nist.gov
Tom Kramer	thomas.kramer@nist.gov
Anthony Pietromartire	pietromartire.anthony@nist.gov
Craig Schlenoff	craig.schlenoff@nist.gov

Contents

List of Figures

1 Introduction

This document describes the different ideas that will need to be implemented to represent and identify action failures during the execution of kitting tasks by a robotic arm. In our kitting domain, PDDL (Planning Domain Definition Language) actions, predicates, and initial and goal states are used by a planner to generate a plan that contains a sequence of PDDL actions to perform in order to build a kit. During the execution of the plan, the interpreter, from the execution module, acquires information on the location of different objects in the workstation.

In order to represent and identify failures during kitting, the following modules need to be implemented:

1. Representation of PDDL actions in the ontology (Section 2)
2. Representation of Failures associated to actions (Section 3)
3. Representation of predicates with state relations (Section 4)
4. Update of the MySQL database (Section 5)

2 Representation of PDDL Actions in the Ontology

Actions are ways of changing the state of the world and consist of a precondition and an effect sections. Predicates and functions constitute preconditions and effects. 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 [1]. Both predicates and functions are well documented in the SVR. Before a robot can perform a PDDL action in the plan, the system needs to validate that the preconditions associated to this action are true. When the action has been carried out by the robot, the system will need to check that the effects for this action are met. In the kitting domain, PDDL actions' preconditions and effects are represented with predicates, negative predicates, and functions.

Figure 1 shows the action *put-part* that will be used as the model to discuss the components of a PDDL action.

```
1. (:action put-part
2.    :parameters(
3.       ?robot - Robot
4.       ?part - Part
5.       ?kit - Kit
6.       ?worktable - WorkTable
7.       ?partstray - PartsTray)
8.    :precondition (and
9.       (part-location-robot ?part ?robot)
10.      (robot-holds-part ?robot ?part)
11.      (on-worktable-kit ?worktable ?kit)
12.      (origin-part ?part ?partstray)
13.      (< (quantity-kit ?kit ?partstray)
14.      (capacity-kit ?kit ?partstray))
15.      (kit-location-worktable ?kit ?worktable))
16.   :effect (and
17.      (not (part-location-robot ?part ?robot))
18.      (not (robot-holds-part ?robot ?part))
19.      (part-not-searched)
20.      (not (found-part ?part ?partstray))
21.      (part-location-kit ?part ?kit)
22.      (increase (quantity-kit ?kit ?partstray) 1)
23.      (robot-empty ?robot))
24. )
```

Figure 1: PDDL action put-part.

1. `action` (line 1): The unique name of the action comes directly after the keyword

`:action`. In this example, the name of the action is `put-part`.

2. `parameters` (lines 2–7): The parameters (preceded by a ? mark) that participate in this action are listed along their types. For example, line 3 can be read as "`robot` is a parameter and is of type `Robot`".

3. `precondition` (lines 8–15): List of all the predicates and functions needed in the precondition section.

4. `effect` (lines 16–23): List of all the predicates and functions needed in the effect section.

The representation of PDDL action in the ontology is made up of the classes Action, Precondition, Effect, Predicate, Function, ParameterList. Operations between functions such as the one shown at lines 13–14 in Figure 1, are expressed with the class FunctionBool. All these classes are subclasses of DataThing.

In the remainder of this section, we provide paragraph descriptions of each of the classes used to represent PDDL actions in the *SOAP* ontology. The naming convention utilized below follows the OWL implementation of the ontology.

1. 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 1, an action has unique name (*hasAction_Name*) of type `string`.

2. ParameterList – The *put-part* action illustrated in Figure 1 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, LargeBoxWithEmptyKitTrays, 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 1, 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.

3. Precondition – A Precondition can consist of only one Predicate (*hasPrecondition_Predicate*), only one Function (*hasPrecondition_Function*), FunctionBool (*hasPrecondition_FunctionBool*), or a combination of these three classes. A Precondition belongs to one Action (*hadByPrecondition_Action*).

4. Effect – An Effect can consist of only one Predicate (*hasEffect_Predicate*), only one Function (*hasEffect_Function*), FunctionBool (*hasEffect_FunctionBool*), or a combination of these three classes. An Effect belongs to one Action (*hadByEffect_Action*). A negative Predicate is represented with the declaration of *hasEffect_Predicate* within the OWL built-in property assertion `owl:NegativePropertyAssertion`.

5. 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.

   The meaning of reference and target parameters lies in the definition of a state variable. We recall the following definition of a state variable $\mathrm{x} : \mathrm{A}_1 \times \cdots \times \mathrm{A}_\mathrm{i} \times \mathrm{S} \rightarrow \mathrm{B}_1 \cup \cdots \cup \mathrm{B}_\mathrm{j}$ $(i, j \geq 1)$ that is used to convert state variables into predicates as follows:

   - $\mathrm{A}_1 \times \cdots \times \mathrm{A}_\mathrm{i} \times \mathrm{S} \rightarrow \mathrm{B}_1 \cup \cdots \cup \mathrm{B}_\mathrm{j}$ $(i, j \geq 1)$
     - predicate_1$(\mathcal{A}, \mathcal{B})$
     - ...
     - predicate_n$(\mathcal{A}, \mathcal{B})$

   Where $\mathcal{A} \in \{\mathrm{A}_1, \ldots, \mathrm{A}_\mathrm{i}\}$ and $\mathcal{B} \in \{\mathrm{B}_1, \ldots, \mathrm{B}_\mathrm{i}\}$ $(i, j \geq 1)$

   From this methodology, we have defined a predicate's parameter as a reference parameter if the parameter belongs to the set $\mathcal{A}$. Similarly, we have defined a predicate's parameter as a target parameter if the parameter belongs to the set $\mathcal{B}$. For instance, the predicate `(part-location-robot ?part ?robot)` has `?part` as the reference parameter and `?robot` as the target parameter. This convention has been used in our ontology to define these two distinct types of parameters.

6. 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.

7. FunctionBool – 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 1 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.

3 Representation of Failures in the Ontology

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 our kitting system, 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 2.

for *each action* $\mathcal{A}$ *in the Plan Instance File* **do**
    Interpreter;
    converts $\mathcal{A}$ into a set $\mathcal{S}$ of Canonical Robot Language commands;
    stores $\mathcal{S}$ in Canonical Robot Language Plan;
end
for *each set* $\mathcal{S}$ *in Canonical Robot Language Plan* **do**
    Robot Controller reads $\mathcal{S}$;
    System Monitor calls Predicate Evaluation;
    Predicate Evaluation;
     traces back action $\mathcal{A}$ from set $\mathcal{S}$;
     computes truth-value of predicates for action $\mathcal{A}$ precondition;
    if *output of Predicate Evaluation is true* **then**
        Robot Controller executes $\mathcal{S}$;
        Predicate Evaluation computes truth-value of predicates for action $\mathcal{A}$ effect;
        if *output of Predicate Evaluation is true* **then**
            end;
        end
        else failure;
    end
    else failure;
end

Figure 2: Failure identification.

As seen in Figure 2, 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 1 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 1: 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`.

4 Representation of Predicates with State Relations

As seen in Section 3, 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 that is redirected back to the System Monitor. We have implemented the concept of "Spatial Relations" in the *SOAP* ontology to be able to compute the truth-value of a predicate.

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

- RCC8_Relation: RCC8 [2] 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 "-" [?]. 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:

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

  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 2. 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:

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

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 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 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”.

5 Update of the MySQL Database

In kitting, the robot moves parts and kit trays in the workstation from an initial state to a goal state in order to build a kit. To guarantee that state relations use current information of objects in the environment (kit trays, parts, etc), it is necessary to update the MySQL database once the locations and orientations of these objects change in the environment. Therefore, the kitting domain needs to include an approach that updates the MySQL database when the configuration of the workstation changes, namely, after a PDDL action has been performed by the robot.

5.1 Proposed Research

We propose to develop a methodology that updates the MySQL database when the locations of objects in the workstation are modified by the execution of an action. The new methodology will identify objects of interest involved in the execution of an action by analyzing the parameters of this action in the PDDL plan file. Once an action is executed by the robot, the location of some of these objects (parameters) will be updated in the MySQL database.
We consider the PDDL action (`take-kittray` *robot_1 kit_tray_1 empty_kit_tray_supply tray_gripper work_table_1*) that is intended to pick up a kit tray (*kit_tray_1*) from a box of empty kit trays (*empty_kit_tray_supply*). In this action, *kit_tray_1* is the object of interest which location will vary during the execution of this action. The proposed methodology will be able to identify *kit_tray_1* as the object of interest for the action `take-kittray` and will thus update the location of *kit_tray_1* in the MySQL database once this action is performed.

References

[1] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*, 20(1):61–124, December 2003.

[2] 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.