Available online at www.sciencedirect.com

ScienceDirect

Robotics and Computer-Integrated Manufacturing 00 (2014) 1–18

Robotics
~~and~~
Computer-
Integrated
~~Manufac~~
turing

Towards Robust Assembly with Knowledge Representation for PDDL

Z. Kootbally[a,*], C. Schlenoff[b], T. Kramer[c], S.K. Gupta[d]

[a] *University of Maryland, College Park, MD 20740, USA*
[b] *Intelligent Systems Division, National Institute of Standards and Technology, Gaithersburg, MD, USA*
[c] *Department of Mechanical Engineering, Catholic University of America, Washington, DC, USA*
[d] *Maryland Robotics Center, University of Maryland, College Park, MD, USA*

Abstract

The effort described in this paper attempts to integrate agility aspects in the ”Agility Performance of Robotic Systems” (APRS) project, developed at the National Institute of Standards and Technology (NIST). The new technical idea for the APRS project is to develop the measurement science in the form of an integrated agility framework enabling manufacturers to assess and assure the agility performance of their robot systems. This framework includes robot agility performance metrics, information models, test methods, and protocols. This paper presents models for the Planning Domain Definition Language (PDDL), used within the APRS project. PDDL is an attempt to standardize Artificial Intelligence planning languages. The described models have been fully defined in the XML Schema Definition Language (XSDL) and in the Web Ontology Language (OWL) for kit building applications. Kit building or kitting is a process that brings parts that will be used in assembly operations together in a kit and then moves the kit to the area where the parts are used in the final assembly. Furthermore, the paper discusses a tool that is capable of automatically and dynamically generating PDDL files from the models in order to replan from scratch or to repair a plan to recover from failures.

Keywords: PDDL (Planning Domain Definition Language), planning, replanning, plan repair, Agility, OWL (Web Ontology Language), knowledge representation, robotics, XSDL (XML Schema Definition Language)

1. Introduction

The new technical idea for the “Agility Performance of Robotic Systems” (APRS) project [1] at the National Institute of Standards and Technology (NIST) is to develop the measurement science in the form of an integrated agility framework enabling manufacturers to assess and assure the agility performance of their robot systems. This framework includes robot agility performance metrics, information models, test methods, and protocols – all of which are validated using a combined virtual and real testing environment. The information models enumerate and make explicit the necessary knowledge for achieving rapid re-tasking and being agile and will answer question such as “What does the robot need to know?”, “When does it need to know it?”, and “How will it get that knowledge?”. This framework will (1) allow manufacturers to easily and rapidly reconfigure and re-task robot systems in assembly operations, (2) make robots more accessible to small and medium organizations, (3) provide large organizations

*Corresponding author: Zeid Kootbally, Department of Mechanical Engineering, University of Maryland, College Park, MD 20740, USA
Email addresses: `zeid.kootbally@nist.gov` (Z. Kootbally), `craig.schlenoff@nist.gov` (C. Schlenoff), `thomas.kramer@nist.gov` (T. Kramer), `skgupta@umd.edu` (S.K. Gupta)

greater efficiency in their assembly operations, and (4) allow the U.S. to compete effectively in the global market. Any company that is currently deploying or planning to deploy robot systems will benefit because it will be able to accurately predict the agility performance of its robot systems and be able to quickly re-task and reconfigure its assembly operations.

The increased number of new models and variants have forced manufacturing firms to meet the demands of a diversified customer base by creating products in a short development cycle, yielding low cost, high quality, and sufficient quantity. Modern manufacturing enterprises have two alternatives to face the aforementioned requirements. The first one is to use manufacturing plants with excess capacity and stock of products in inventory to smooth fluctuations in demand. The second one is to use and increase the flexibility of their manufacturing plants to deal with the production volume and variety. While the use of flexibility generates the complexity of its implementation, it still is the preferred solution. Chryssolouris [2] identified manufacturing flexibility as an important attribute to overcome the increased number of new models and variants from customized demands. Flexibility, however needs to be defined in a quantified fashion before being considered in the decision making process.

Agility is often perceived as combination of speed and flexibility. Gunasekaran [3] defines agile manufacturing as the capability to survive and prosper in a competitive environment of continuous and unpredictable change by reacting quickly and effectively to changing markets, driven by customer-designed products and services. To be able to respond effectively to changing customer needs in a volatile marketplace means being able to handle variety and introduce new products quickly. Lindbergh [4] and Sharafi & Zhang [5] mentioned that agility consists of flexibility and speed. Essentially, an organisation must be able to *respond flexibly* and *respond speedily* [6]. Conboy & Fitzgerald [7] identified terms such as *speed* [8], *quick* [9, 10, 11, 12], *rapid* [13], and *fast* [14] that occur in most definitions of agility.

The above definitions of agile manufacturing can be applied at the assembly level of a manufacturing system. The assembly system needs to have a certain level of flexibility in the presence of disturbances that can be expressed by the degree of robustness. Kannan & Parker [15] described robustness as the ability of the system to identify and recover from faults. Robustness of a control system was described by Leitão [16] as the capability to remain working correctly and relatively stable, even in presence of disturbances. The concept of robustness discussed in this paper is expressed with replanning and plan repair for failure recovery (e.g., misalignments, incorrect parts and tooling, shortage of parts, or missing tool). Fox *et al.* [17] discussed replanning and plan repair when differences are detected between the expected and actual context of execution during plan execution in real environments. The latter authors define plan repair as the work of adapting an existing plan to a new context while perturbing the original plan as little as possible. Replanning is defined as the work of generating a new plan from scratch.

This paper first describes the models developed to represent structures of the planning language in the APRS project. The APRS project is working in collaboration with the IEEE Robotics and Automation Society's Ontologies for Robotics and Automation (ORA) Working Group to develop information models related to kitting, including a model of the kitting environment and a model of a kitting plan. Kitting or kit building is a process that brings parts that will be used in assembly operations together in a kit and then moves the kit to the area where the parts are used in the final assembly. It is anticipated that utilization of the knowledge representation will allow for the development of higher performing kitting systems and will lead to the development of agile automated robot assembly.

Planning for kitting relies on the Planning Domain Definition Language (PDDL) [18]. 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. Structures of PDDL domain and problem files are fully defined in each of two languages: XML Schema Definition Language (XSDL) [19, 20, 21] and Web Ontology Language (OWL) [22, 23, 24]. Furthermore, this paper describes a tool that is capable of automatically and dynamically generating PDDL domain and problem files from the OWL models.

This paper is structured as follows: An overview of the knowledge driven methodology for the APRS project is presented in Section 2. The XSDL models that were developed to represent PDDL domain and problem files are discussed in Section 3. A tool that is capable of automatically and dynamically producing PDDL domain and problem files from OWL files is described in Section 4. Finally, concluding remarks and future work are addressed in Section 5.

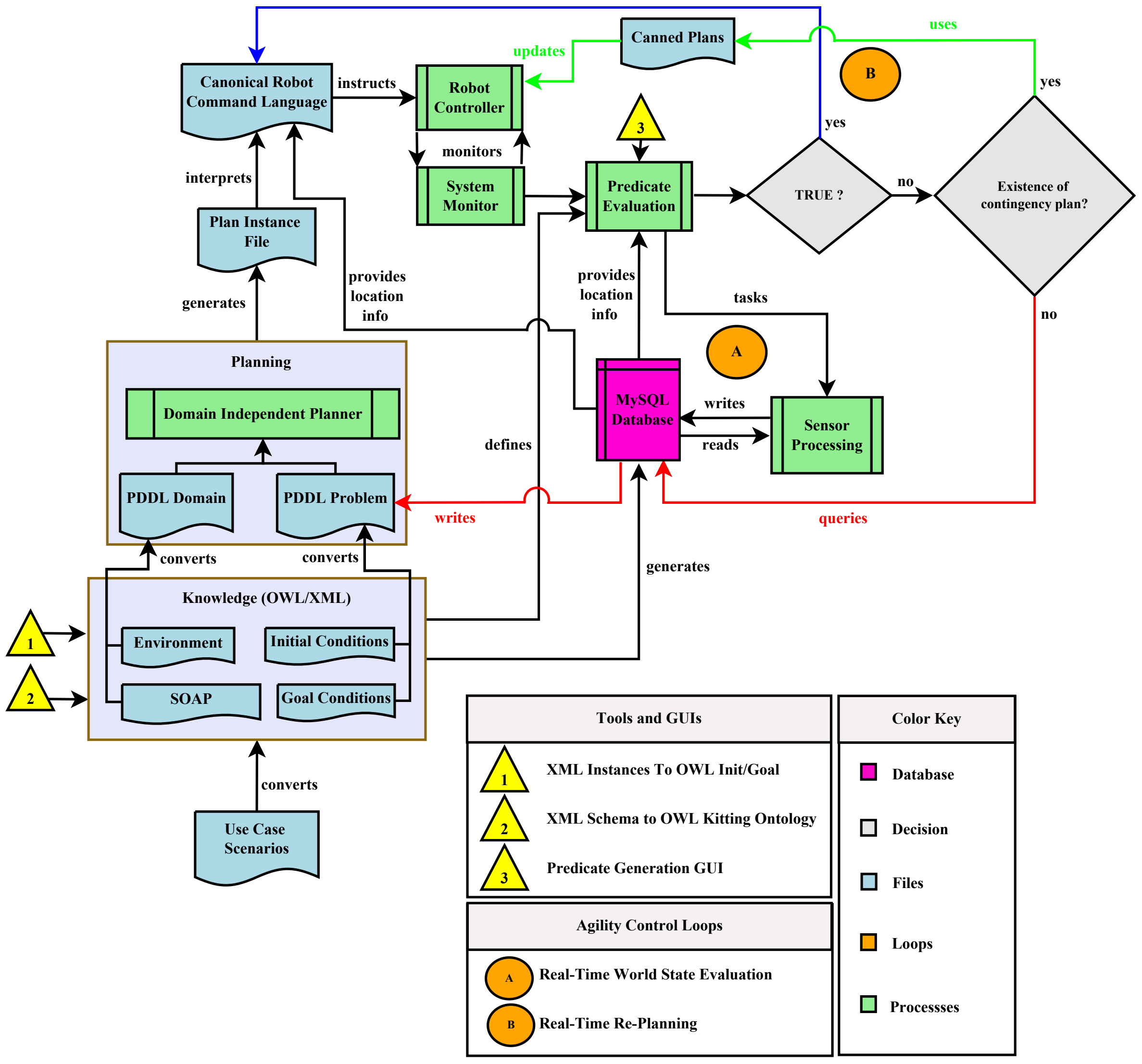


Figure 1: Knowledge Driven Design extensions.

2. Knowledge Driven Methodology

The knowledge driven methodology presented in this section is not intended to act as a stand-alone system architecture. Rather it is intended to be an extension to well developed hierarchical, deliberative architectures such as 4D/RCS [25]. The overall knowledge driven methodology of the system is depicted in Figure 1. The remainder of this section gives a brief description of the components pertaining to the effort presented in this paper.

- Use Case Scenarios – At the early stage of assembly, new orders coming from customers are entered in the system by an operator via a graphical user interface. The information that is required in this step is for instance the type of assembly and the number of products required. This first step is therefore an attempt to introduce agility in the system with a functionality that smooths fluctuations in demand.

- Knowledge (OWL/XML) – At the next level up, the information encoded in the Use Case Scenarios is then organized into a domain independent representation. The Knowledge (OWL/XML) component contains all of the basic information that was determined to be needed during the evaluation of the Use Case Scenarios. This component consists of class files and instance files that describe the environment (Environment), including the initial (Initial Conditions) and goal (Goal Conditions) states for the current assembly, and PDDL actions (SOAP). The knowledge is represented in a compact form with knowledge classes inheriting common attributes from parent classes. The SOAP knowledge describes aspects of PDDL actions that are required for the domain under study. The instance files describe the initial and goal states for the system through the Initial Conditions file and the Goal Conditions file, respectively. 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. The goal state file only needs to contain information that is relevant to the end goal of the system.

  Since both the OWL and XML implementations of the knowledge representation are file based, real time information proved to be problematic. In order to solve this problem, an automatically generated MySQL database has been introduced as part of the knowledge representation.

- Planning – At the next level up, aspects of this knowledge are automatically extracted and encoded in a form that is optimized for a planning system to utilize. The planning language used in the knowledge driven system is PDDL. 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 a set of OWL files. The PDDL Domain file is produced from the Environment and the SOAP OWL files while the PDDL Problem file is produced from the Initial Condition and the Goal Condition files. From the PDDL Domain and PDDL Problem files, a domain independent planning system [26] was used to produce a static Plan Instance File.

- Canonical Robot Command Language – Once a PDDL plan has been formulated, the knowledge is transformed into a representation that is optimized for use by a robotic system. The interpreter combines knowledge from the PDDL plan with knowledge from the MySQL database to form a sequence of low level commands 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.

- Robot Controller – CRCL commands are then sent to the Robot Controller. One PDDL action from the Plan Instance File is interpreted into a set of CRCL commands. Each set of CRCL commands is queued and the oldest entries are processed first (FIFO).

- Predicate Evaluation – The Predicate Evaluation process is used to check if the preconditions and effects for a PDDL action are satisfied [27]. This process intrinsically identifies failures during the execution of an action by the robot. Each precondition and each effect is a predicate expression that must be respectively validated before and after an action is performed. The world model (the MySQL database) is queried for the pose and class of each relevant parameter for a given 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.

  Two distinct results come out of the Predicate Evaluation process.

  1. All predicates within the precondition and effect sections for the current action are true. In this case, the next set of CRCL commands are performed (blue arrow).
  2. At least one predicate within the precondition or effect section is false. This case is considered a failure and two situations are checked. The system provides various known failure modes that could exist for the combination of predicates that were found deficient. It provides the consequences of such a failure occurring, remedial information for such failure, and the chance that this kind of failure could occur. In

the case a failure mode is provided for the current failure, Canned Plans are used for failure remediation (green arrows). When no failure modes exist for the failed predicate(s), replanning or plan repair is performed (red arrows). Before replanning or plan repair takes place, the MySQL database is queried in order to build the initial state of the environment in the PDDL problem file. This ensures that the initial state of the environment is properly set with current information that will be used to generate a new plan.

3. Models for PDDL Domain and Problem

This section describes XSDL models that were developed to represent PDDL structures for domain and problem files. In this project, a two-step process is required to generate PDDL files: (1) XSDL files and XML instance files are used to generate a set of OWL files, (2) the generated OWL files are then used to produce the PDDL files.

The reader may ask about the necessity of the first step and may find it odd that the PDDL files are not directly encoded in OWL by a human expert. Moreover, the reader may ask about the necessity of using OWL as an intermediate step to generate PDDL files and why not directly going from the XSDL models to PDDL. As mentioned in the introductory section, the APRS project is working in collaboration with the ORA Working Group to develop information models related to kitting. Early in its existence, the ORA Working Group made a commitment to use OWL for its models. As the authors used OWL, difficulties arose as summarized in [28]. The models being built lent themselves to a more structured object model approach of the sort used in languages such as EXPRESS [29], C++ classes [30], and XSDL. It was decided to use XSDL as the language for initial modeling in the APRS project and to produce OWL models from the XSDL models. Moreover, one author already had experience with XSDL and was building C++ software tools for manipulating XML schemas and instance files. To make the translation work easier and more reliable, additional C++ tools were built for that purpose.

3.1. PDDL Background and Structure

Since its first release in 1998 as the problem-specific language for the AIPS-98 planning competition [31], PDDL has become a community standard for the representation and exchange of planning domain models. Although the early days of PDDL showed some dissatisfaction in the community, considerable improvements were made to the language, thus enabling the comparison between systems sharing the standard and increasing the availability of shared planning resources. The introduction of PDDL has facilitated the scientific development of planning [18].

PDDL 2.1 is used for the effort presented in this paper. PDDL 2.1 offers a revised version from the original version of the syntax for expressing numeric-valued fluents. Gerevini *et al.* [32] define a numeric fluent as a state variable over the set $\mathbb{R}$ of real numbers such that there exists at least one domain action that can change its initial value specified in the problem initial state. Fox & Long [18] proposed a definitive syntax for the expression of numeric fluents. The authors provided some minor revisions to the version proposed by McDermott [33]. Another feature introduced in PDDL 2.1 as an optional field within the specification of problems is a plan metric. Plan metrics specify the basis on which a plan will be evaluated for a given problem. Different optimal plans can be produced with different plan metrics for the same initial and goal states. The use of PDDL 2.1 for the effort presented in this paper was motivated by numeric fluents and plan metrics. Even though the plan metrics feature is not currently used in this effort, it is the intention of the authors to do so as the project grows.

3.1.1. PDDL Domain File

The development of XSDL models for PDDL requires the analysis of PDDL domain and problem files structures. Figure 2 is an excerpt of the PDDL domain file created for kitting. This excerpt is used only for the purpose of this paper. The complete PDDL domain file for kitting consists of 12 `types`, 34 `predicates`, 9 `functions`, and 10 `actions`. The structure of a PDDL domain file is separated in sections that are described below:

- line 1: The keyword `domain` signals a planner that this file contains information on the domain. `kitting-domain` is the name given to the domain in the example.
- line 2: It can be seen in the example that PDDL includes a syntactic representation of the level of expressivity required in particular domain descriptions through the use of `requirements` flags. This gives the opportunity for a planning system to reject attempts to plan with domains that make use of more advanced features of the language than the planner can handle.

```
 1 (define (domain kitting-domain)
 2     (:requirements :strips :typing :derived-predicates :action-costs :fluents :equality)
 3     (:types
 4         EndEffector EndEffectorHolder Kit KitTray
 5         LargeBoxWithEmptyKitTrays LargeBoxWithKits
 6         Part PartsTray EndEffectorChangingStation
 7         Robot StockKeepingUnit WorkTable)
 8     (:predicates
 9         (endEffector-has-no-heldObject ?endeffector - EndEffector)
10         (endEffector-is-for-kitTraySKU ?endeffector - EndEffector ?sku - StockKeepingUnit)
11         (endEffector-has-physicalLocation-refObject-robot ?endeffector - EndEffector ?robot - Robot)
12         (kitTray-has-skuObject-sku ?kittray - KitTray ?sku - StockKeepingUnit)
13         (kitTray-has-physicalLocation-refObject-lbwekt ?kittray - KitTray ?lbwekt - LargeBoxWithEmptyKitTrays)
14         (robot-has-endEffector ?robot - Robot ?endeffector - EndEffector)
15         (endEffector-has-heldObject-kitTray ?endeffector - EndEffector ?kittray - KitTray)
16         (kitTray-has-physicalLocation-refObject-endEffector ?kittray - KitTray ?endeffector - EndEffector))
17     (:functions
18         (quantity-of-parts-in-partstray ?partstray - PartsTray)
19         (quantity-of-parts-in-kit ?sku - StockKeepingUnit ?kit - Kit)
20         (quantity-of-kittrays-in-lbwekt ?lbwekt - LargeBoxWithEmptyKitTrays)
21         (quantity-of-kits-in-lbwk ?lbwk - LargeBoxWithKits)
22         (current-quantity-of-parts-in-kit ?kit - Kit)
23         (final-quantity-of-parts-in-kit ?kit - Kit)
24         (capacity-of-parts-in-kit ?partsku - StockKeepingUnit ?kit - Kit)
25         (capacity-of-kits-in-lbwk ?lbwk - LargeBoxWithKits)
26         (part-found-flag))
27     (:action take-kitTray
28         :parameters(
29             ?robot - Robot
30             ?kittray - KitTray
31             ?lbwekt - LargeBoxWithEmptyKitTrays
32             ?endeffector - EndEffector
33             ?sku - StockKeepingUnit)
34         :precondition(and
35             (> (quantity-of-kittrays-in-lbwekt ?lbwekt) 0)
36             (endEffector-has-no-heldObject ?endeffector)
37             (endEffector-is-for-kitTraySKU ?endeffector ?sku)
38             (endEffector-has-physicalLocation-refObject-robot ?endeffector ?robot)
39             (kitTray-has-skuObject-sku ?kittray ?sku)
40             (kitTray-has-physicalLocation-refObject-lbwekt ?kittray ?lbwekt)
41             (robot-has-endEffector ?robot ?endeffector))
42         :effect(and
43             (decrease (quantity-of-kittrays-in-lbwekt ?lbwekt) 1)
44             (endEffector-has-heldObject-kitTray ?endeffector ?kittray)
45             (kitTray-has-physicalLocation-refObject-endEffector ?kittray ?endeffector)
46             (not (endEffector-has-no-heldObject ?endeffector))
47             (not (kitTray-has-physicalLocation-refObject-lbwekt ?kittray ?lbwekt))))
48 )
```

Figure 2: Excerpt of the PDDL domain file for kitting.

- lines 3–7: Object types have to be declared before they are used in `predicates` and `functions`. This is done with the declaration `(:types name`$_1$ `...` `name`$_n$`)`.
- lines 8–16: The `predicates` part of a domain definition specify only what are the predicate names used in the domain, and their number of arguments (and argument types, if the domain uses typing). The "meaning" of a predicate, in the sense of for what combinations of arguments it can be true and its relationship to other predicates, is determined by the effects that actions in the domain can have on the predicate, and by what instances of the predicate are listed as true in the initial state of the problem definition.
- lines 17–26: functions are used to declare numeric fluents. Numeric assignments (initial value of each function) are set in the initial state of the problem file and change when an action is executed. The declaration of functions is similar to predicates.
- lines 27–47: The domain is described in terms of action schemata. An action schemata specifies a way that executing an action affects the state of the world. An action schemata includes `parameters`, `preconditions`, and `effects`. An `action` is identified by a unique name (`take-kitTray` at line 27). Each `parameter` of an `action` is defined by a name and a type (e.g., at line 29, `robot` is the name of the parameter and `Robot` is its

type). `Preconditions` and `effects` may consist of positive predicates (lines 36–41 and lines 44–45). Only preconditions may contain conditions on numeric expressions (line 35). Conditions on numeric expressions are always comparisons between pairs of numeric expressions. They include comparisons between a function and a number (a positive integer) or comparisons between two functions. Only effects may contain function operations (line 43) and negative predicates (lines 46–47). Function operations are used to update the values of primitive numeric expressions.

3.1.2. PDDL Problem File

A problem is what a planning system tries to solve. A problem specifies an initial situation and a goal to be achieved. Figure 3 is a portion of the PDDL problem file created for kitting. This excerpt shows the different components of a generic PDDL problem file. These components are described below.

```
 1 (define (problem kitting-problem)
 2     (:domain kitting-domain)
 3     (:objects
 4         robot_1 - Robot
 5         changing_station_1 - EndEffectorChangingStation
 6         kit_tray_1 - KitTray
 7         kit_1 - Kit
 8         empty_kit_tray_supply - LargeBoxWithEmptyKitTrays
 9         finished_kit_receiver - LargeBoxWithKits
10         work_table_1 - WorkTable
11         part_a_tray part_b_tray - PartsTray
12         part_a_1 part_b_1 - Part
13         part_gripper tray_gripper - EndEffector
14         part_gripper_holder tray_gripper_holder - EndEffectorHolder
15         stock_keeping_unit_part_a stock_keeping_unit_part_b
16         stock_keeping_unit_kit_tray - StockKeepingUnit)
17     (:init
18         (endEffector-has-no-heldObject  part_gripper)
19         (endEffector-has-no-heldObject  tray_gripper)
20         (endEffector-is-for-kitTraySKU tray_gripper stock_keeping_unit_kit_tray)
21         (endEffector-is-for-partSKU part_gripper stock_keeping_unit_part_a)
22         (endEffector-is-for-partSKU part_gripper stock_keeping_unit_part_b)
23         (endEffectorHolder-has-physicalLocation-refObject-changingStation part_gripper_holder changing_station_1)
24         (endEffectorHolder-has-physicalLocation-refObject-changingStation tray_gripper_holder changing_station_1)
25         (kitTray-has-skuObject-sku kit_tray_1 stock_keeping_unit_kit_tray)
26         (partsVessel-has-part part_a_tray part_a_1)
27         (partsVessel-has-part part_b_tray part_b_1)
28 
29         (= (quantity-of-parts-in-partstray part_a_tray) 1)
30         (= (quantity-of-parts-in-partstray part_b_tray) 1)
31         (= (quantity-of-parts-in-kit stock_keeping_unit_part_a kit_1) 0)
32         (= (quantity-of-parts-in-kit stock_keeping_unit_part_b kit_1) 0)
33         (= (quantity-of-kittrays-in-lbwekt empty_kit_tray_supply) 1)
34         (= (quantity-of-kits-in-lbwk finished_kit_receiver) 0)
35         (= (capacity-of-parts-in-kit stock_keeping_unit_part_a kit_1) 1)
36         (= (capacity-of-parts-in-kit stock_keeping_unit_part_b kit_1) 1)
37         (= (capacity-of-kits-in-lbwk finished_kit_receiver) 12)
38         (= (current-quantity-of-parts-in-kit kit_1) 0)
39         (= (final-quantity-of-parts-in-kit kit_1) 2)
40         (= (part-found-flag) 1))
41     (:goal(and
42         (kit-has-physicalLocation-refObject-lbwk kit_1 finished_kit_receiver)
43         (lbwk-has-kit finished_kit_receiver kit_1)))
44 )
```

Figure 3: The PDDL problem file for kitting.

- line 1: The keyword `problem` signals a planner that this file contains information on the problem. `kitting-problem` is the name given to the problem in the example.
- line 2: A problem is defined with respect to a domain. The keyword `domain` is a reference to the domain to which the problem is associated. In the example, the problem is defined with respect to the domain described in Figure 2.

- lines 3–16: `objects` specifies the distinct instances and types of objects that will appear in the initial and goal states.

- lines 17–40: The `init` section consists of predicates that are true in the initial state. Because of the closed world assumption of PDDL, predicates not specified in the `init` section are set to false. The initial value of each `function` described in the domain is set in the `init` section. For instance, line 29 tells the planning system that part_a_tray contains 1 part in the initial state. In Figure 3, function assignments are depicted at lines 29–40.

- lines 41–43: The `goal` section specifies the predicates that need to be true in the goal state. The value that a `function` needs to reach may also be specified in the `goal` section.

3.2. Models for PDDL Domain

A closer look at Figure 3 shows that all the basic components (`objects`, `predicates`, and `functions`) in the problem are also defined in the domain. The only difference is that the problem requires instance parameters while the domain uses generic parameters. Therefore, only the PDDL domain needs to be modeled and the information that goes in the definition of the problem can be mapped in the program.

The authors have modeled the structure of a PDDL domain in the SOAP schema. SOAP stands for States, Ordering constructs, Actions, and Predicates. To remove any confusion on this acronym, the authors need to clarify that models of states, ordering constructs, actions, and predicates were used in a sister project, however, only actions and predicates from the SOAP schema are used in the APRS project.

The `types` section in the domain and the `objects` section in the problem contain information that is stored in the KittingWorkstation model. This model is imported by the SOAP model. SolidObject and DataThing constitute the two top-level classes of the KittingWorkstation ontology model, from which all other classes are derived. SolidObject models solid objects, things made of matter. The KittingWorkstation ontology includes several subclasses of SolidObject that are formed from components that are SolidObject. The DataThing class models data for SolidObject. The KittingWorkstation model is fully documented in [28].

To describe models of PDDL domains in SOAP, the authors will often refer to Figure 2. The different figures of classes presented in the remainder of this section were generated by XMLSpy [34]. In these figures, a dotted line around a box means the attribute is optional (may occur zero times), a $0..\infty$ underneath a box means it may not occur, with no upper limit on the number of occurrences, and a $1..\infty$ underneath a box means it may occur at least once, with no upper limit on the number of occurrences. Moreover, the following conventions are adopted in the model descriptions:

- Elements with the suffix Type: An element in the model with the suffix Type is either a XML schema simpleType or a complexType. The simpleType element defines a simple type and specifies the constraints and information about the values of attributes or text-only elements. The complexType element defines a complex type. A complex type element is an XML element that contains other elements and/or attributes. simpleType and complexType elements are translated into OWL classes.

- Elements with the suffix *Name*: An element in the model with the suffix *Name* designates that the element refers to either a simpleType or a complexType. Referencing a simpleType/complexType requires that the simpleType/complexType is already defined.

Domain: A PDDL domain is modeled with DomainType. DomainType extends DataThingType and consists of a name (inherited), a set of requirements, a set of variables, a set of predicates, an optional set of functions, and a set of actions. These components model a PDDL domain file such as the one shown in Figure 2. Components of DomainType are described below.

- *Requirement* and *Variable*: *Requirement* and *Variable* respectively represent the `requirements` and the `types` sections in the PDDL domain file. Since the term Type is already used to designate a XSDL type, the PDDL term `types` was replaced by *Variable*. A *Requirement* and a *Variable* are of type xs:NMTOKEN in XSDL and of type string in OWL.

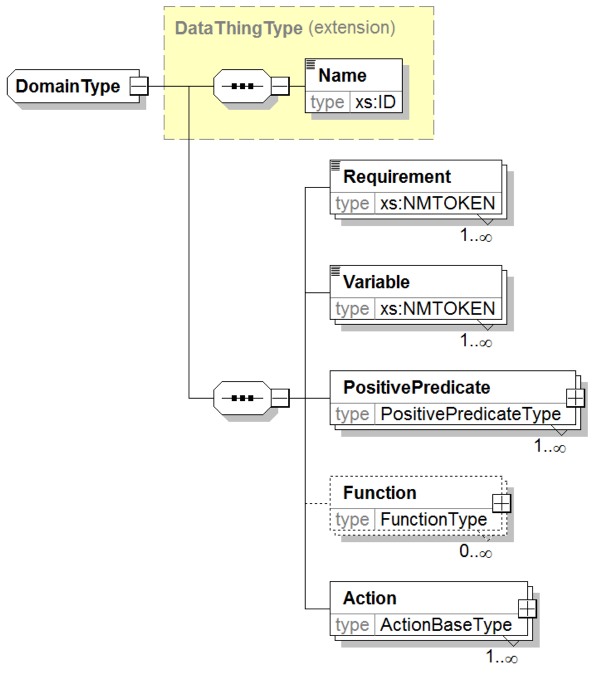


Figure 4: DomainType for modeling PDDL domain files.

- *PositivePredicate*: PDDL predicates are used in positive and negative forms. A PDDL positive predicate is modeled with PositivePredicateType, which is depicted in Figure 5. PositivePredicateType extends DataThingType and consists of a name (inherited), an optional description, a reference parameter, and an optional target parameter. The predicates used in the kitting PDDL domain and problem files all have at least one parameter and can have up to two parameters. In the case a predicate has two parameters, the first parameter is identified as the *ReferenceParameter* and the second parameter is identified as the *TargetParameter*. In the case a predicate has only one parameter, this parameter is identified as the *ReferenceParameter*.

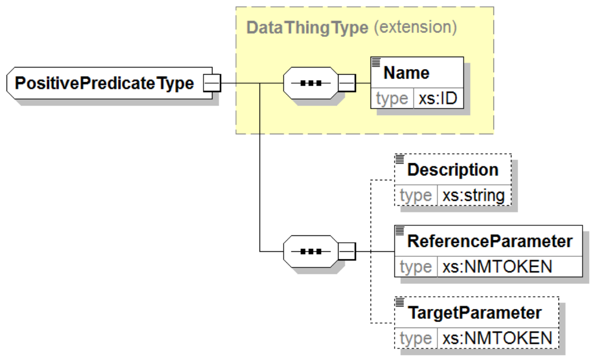


Figure 5: PositivePredicateType for modeling PDDL positive predicates.

- *Function*: PDDL functions are modeled with FunctionType, which is depicted in Figure 6. FunctionType extends DataThingType and consists of a name (inherited), an optional description, an optional reference parameter, and an optional target parameter. As one can note, the reference and target parameters are both optional for a FunctionType since some PDDL functions in our kitting domain are void of parameters such as `(part-found-flag)` at line 26 in Figure 2.

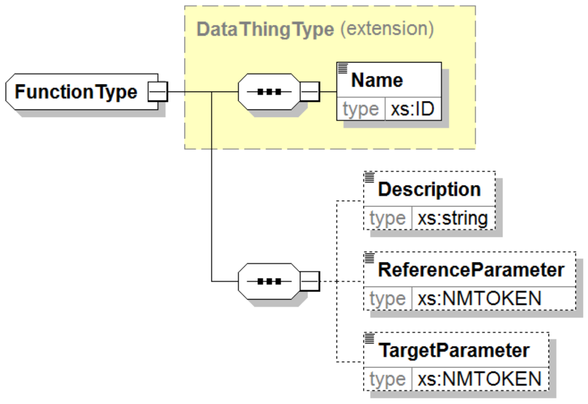


Figure 6: FunctionType for modeling PDDL functions.

- *Action*: PDDL actions are modeled with ActionBaseType, which is depicted in Figure 7. ActionBaseType extends DataThingType. ActionBaseType consists of a unique name (inherited), an optional description, a set of parameters, a precondition section, and an effect section. The components of ActionBaseType are described as follows:

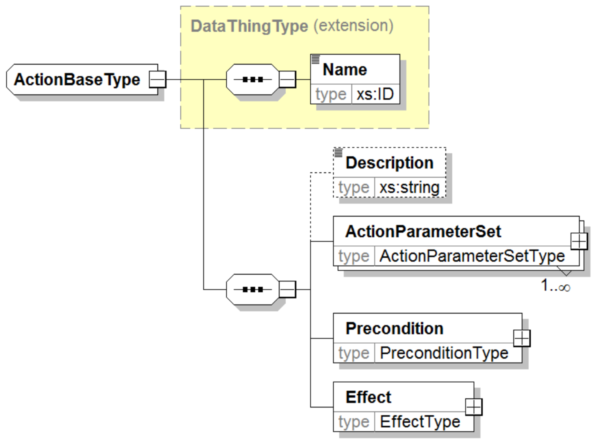


Figure 7: ActionBaseType for modeling PDDL actions.

  - *Name*: The unique name of a PDDL action is assigned with the inherited name attribute.
  - *Description*: A description is written in the generated PDDL file as a PDDL comment. The purpose of a description is only to inform the user about the role of a PDDL action.
  - *ActionParameterSet*: PDDL actions' parameters are modeled with ActionParameterSetType (see Figure 8). ActionParameterSetType consists of a unique name (inherited), the type of the parameter, which is identified with *ActionParameter*, and the position of the parameter, identified with *ActionParameterPosition*, in the list of parameters for a PDDL action [1]. To illustrate these components, the reader may refer

[1]The authors are currently using numbers (integers) to represent orders of parameters in a list of parameters as no built-in structure exists for the representation of ordered lists in OWL.

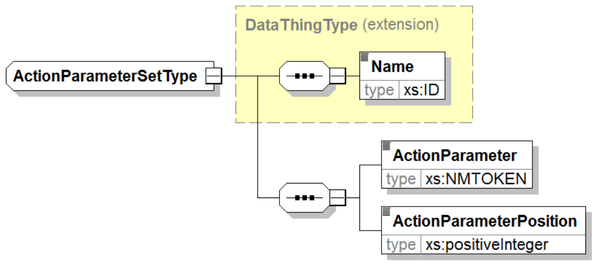


Figure 8: ActionParameterSetType for modeling PDDL actions' parameters.

to the action `take-kitTray` at line 29 in Figure 2. The action `take-kitTray` consists of five parameters where each parameter is an ActionParameterSetType. The *ActionParameter* for the parameter `robot` is `Robot` and its *ActionParameterPosition* is 1.

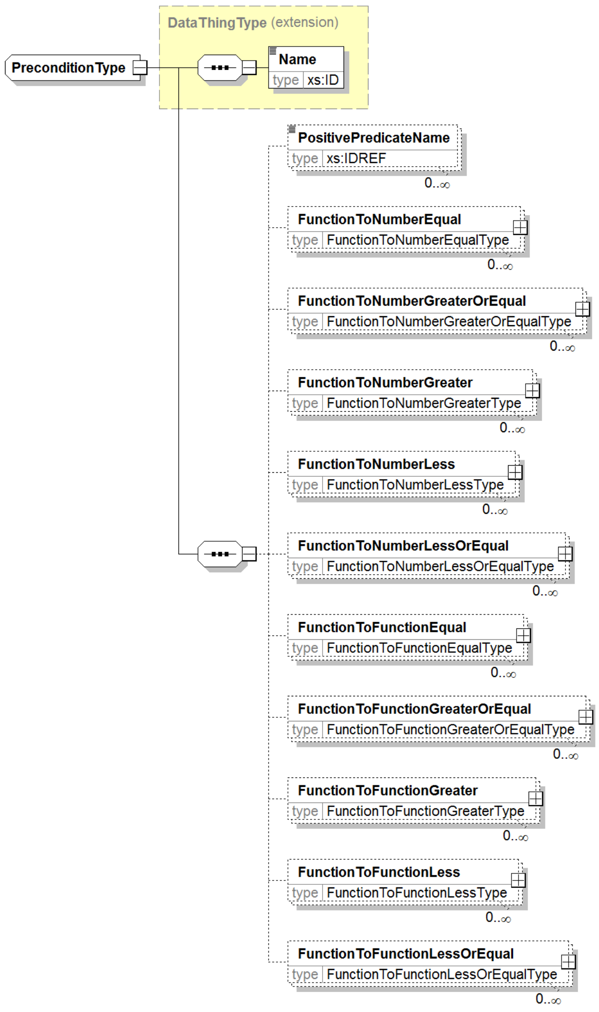

DataThingType (extension)
PreconditionType
Name
type xs:ID
PositivePredicateName
type xs:IDREF
0..∞
FunctionToNumberEqual
type FunctionToNumberEqualType
0..∞
FunctionToNumberGreaterOrEqual
type FunctionToNumberGreaterOrEqualType
0..∞
FunctionToNumberGreater
type FunctionToNumberGreaterType
0..∞
FunctionToNumberLess
type FunctionToNumberLessType
0..∞
FunctionToNumberLessOrEqual
type FunctionToNumberLessOrEqualType
0..∞
FunctionToFunctionEqual
type FunctionToFunctionEqualType
0..∞
FunctionToFunctionGreaterOrEqual
type FunctionToFunctionGreaterOrEqualType
0..∞
FunctionToFunctionGreater
type FunctionToFunctionGreaterType
0..∞
FunctionToFunctionLess
type FunctionToFunctionLessType
0..∞
FunctionToFunctionLessOrEqual
type FunctionToFunctionLessOrEqualType
0..∞

Figure 9: PreconditionType for modeling PDDL actions' preconditions.

– *Precondition*: The precondition section of a PDDL action is modeled with PreconditionType, which is depicted in Figure 9. A PreconditionType extends DataThingType. A PreconditionType consists of a unique name (inherited), optional references to positive predicates, and optional references to conditions on functions. Components of PreconditionType are summarized below.

∗ *Name*: The unique name of a PDDL action's precondition is assigned with the inherited name attribute.

∗ *PositivePredicateName*: A PositivePredicateName refers to a PositivePredicateType, meaning that specific PositivePredicateTypes need to be declared before they are referenced.

∗ Conditions on functions: Conditions on functions are modeled with FunctionConditionType. A FunctionConditionType extends DataThingType and is used to compare two FunctionTypes with each other or to compare a FunctionType with a number. Comparisons between two FunctionTypes are modeled with FunctionToFunctionConditionType. Comparisons between a FunctionType and a number are modeled with FunctionToNumberConditionType. More information on FunctionToFunctionConditionType and on FunctionToNumberConditionType is given below.

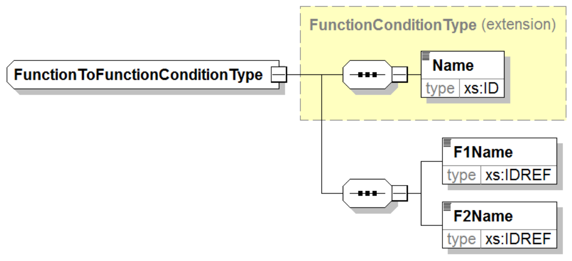

FunctionConditionType (extension)
FunctionToFunctionConditionType
Name
type xs:ID
F1Name
type xs:IDREF
F2Name
type xs:IDREF

Figure 10: FunctionToFunctionConditionType for comparing two PDDL functions.

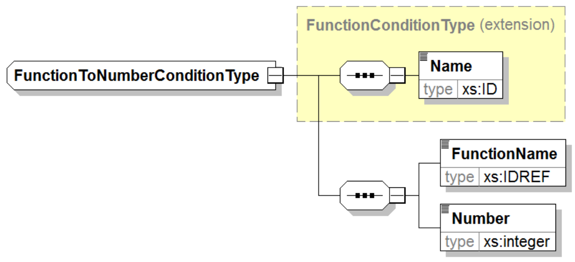

FunctionConditionType (extension)
FunctionToNumberConditionType
Name
type xs:ID
FunctionName
type xs:IDREF
Number
type xs:integer

Figure 11: FunctionToNumberConditionType for comparing PDDL functions with numbers.

· A FunctionToFunctionConditionType extends FunctionConditionType (see Figure 10) and consists of a name (inherited), a reference to the first FunctionType (identified with *F1Name*), and a reference to the second FunctionType (identified with *F2Name*). Comparisons between FunctionTypes require definitions of mathematical symbols ("<", "≤", "=", "≥", and '>"), which are expressed with subtypes of FunctionToFunctionConditionType. The mapping between mathematical symbols and subtypes of FunctionToFunctionConditionType is performed as follows: "<" is modeled with FunctionToFunctionLessType, "≤" is modeled with FunctionToFunctionLessOrEqualType, "=" is modeled with FunctionToFunctionEqualType, "≥" is modeled with FunctionToFunctionGreaterOrEqualType, and ">" is modeled with FunctionToFunctionGreaterType.

· A FunctionToNumberConditionType extends FunctionConditionType (see Figure 11) and consists of a name (inherited), a reference to a FunctionType (identified with *FunctionName*), and a number (identified with *Number*). Similar to FunctionToFunctionConditionType, subtypes

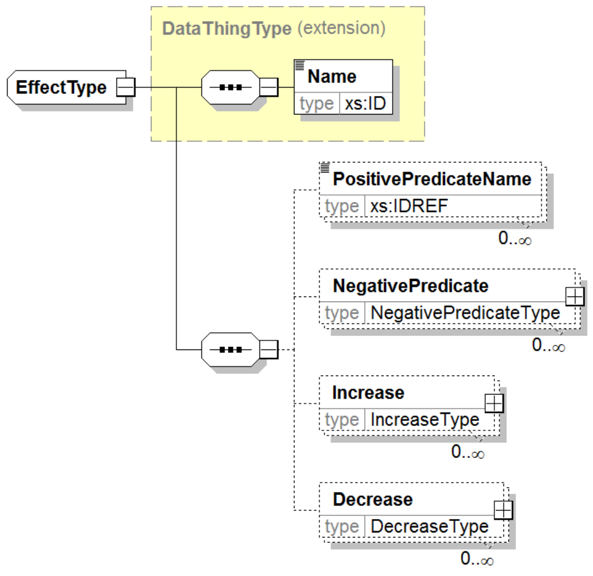


Figure 12: EffectType for modeling PDDL actions' effects.

of FunctionToNumberConditionType indicates the mathematical symbols used for the comparison between a FunctionType and a number. The mapping between mathematical symbols and subtypes of FunctionToNumberConditionType is performed as follows: "<" is modeled with FunctionToNumberLessType, "≤" is modeled with FunctionToNumberLessOrEqualType, "=" is modeled with FunctionToNumberEqualType), "≥" is modeled with FunctionToNumberGreaterOrEqualType, and ">" is modeled with FunctionToNumberGreaterType. An illustration of a FunctionToNumberGreaterType is given at line 35 in Figure 2.

– *Effect*: The effect section of a PDDL action is modeled with EffectType (see Figure 12). EffectType extends DataThingType. EffectType consists of a unique name (inherited), optional references to positive predicates, optional definitions of negative predicates, and optional references to function operations (identified with *Increase* and *Decrease*). Information on each component of EffectType is given below.

  ∗ *PositivePredicateName* is a reference to a positive predicate. This requires that the referenced PositivePredicateType is already defined.

  ∗ *NegativePredicate* describes a NegativePredicateType (see Figure 13). A NegativePredicateType extends DataThingType and models the negation of a predicate. A NegativePredicateType consists of a name (inherited) and a reference to a positive predicate.

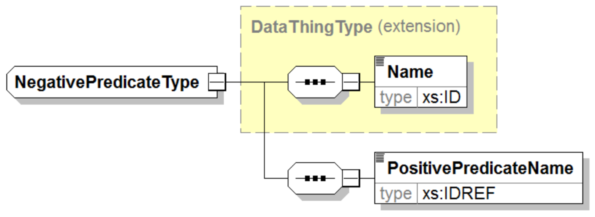
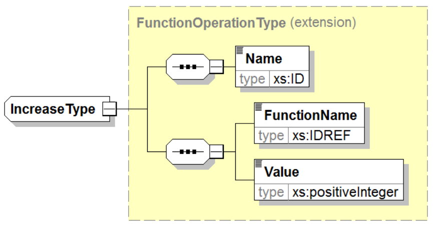
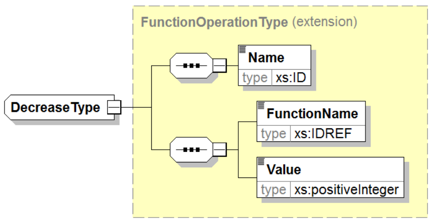


Figure 13: NegativePredicateType for modeling the negation of PDDL positive predicates.

  ∗ Function operations are arithmetic expressions on a FunctionType. Effects can make use of a selection of assignment operations in order to update the values of primitive numeric expressions. The value of a function can be decreased or increased by a certain amount. Function operations are modeled with FunctionOperationType which extends DataThingType. Subtypes of FunctionOperationType are IncreaseType (see Figure 14a) and DecreaseType (see Figure 14b). IncreaseType and DecreaseType consist of a unique name (inherited), a reference to a function (identified with *FunctionName*), and a value (identified by *Value*) by which the function is increased or decreased, respectively. An instance of DecreaseType can be found at line 43 in Figure 2 where the function is `(quantity-of-kittrays-in-lbwekt ?lbwekt)` and the value is `1`.



(a) IncreaseType.



(b) DecreaseType.

Figure 14: Subtypes of FunctionOperationType.

4. Automatic Generation of PDDL Files

Once the schema models for PDDL were in place, an XML instance file that conforms to the schema models was developed. Under the XML standards, an XML data file conforming to an XML schema must be in a different format than the schema and must contain different sorts of statements. An XML statement naming the XML schema file to which an instance file corresponds is normally given near the beginning of the instance file. Many different instance files may correspond to the same schema. The form of an XML instance file is a tree in which instances of the elements of each type are textually inside the instance of the type. Schema models and the XML instance file are used by a set of tools to automatically generate a set of OWL files. The tools required to perform this mechanism were developed by one of the authors at NIST. More information about this set of tools can be found in [28].

To date, the PDDL domain and problem files are hand generated. An expert needs to write these two files and it takes a considerable amount of time to complete these files. A Java-based tool was developed in this effort to automatically and dynamically build these PDDL files. The tool was developed in Java because of its inherent ability to interface with OWL API [35]. The OWL API is a Java API and reference implementation for creating, manipulating and serialising OWL Ontologies. Please note that the use of the language and the API is the authors' choice and the same result may be obtained differently.

4.1. Automatic Generation of the Domain File

The generation of the PDDL domain file is performed by reading the OWL classes from the SOAP OWL file. Since a PDDL domain file stays unchanged during a replanning process, the blueprint of the PDDL domain file is programmed. The tool only needs to access each part of this blueprint from the ontology and outputs this information in a PDDL domain file. Figure 15 shows the components that are read from the SOAP OWL file, stored in the program, and written in the PDDL domain file.

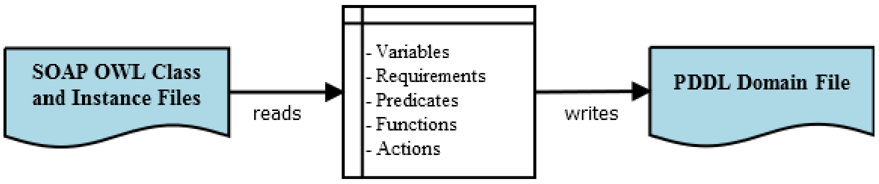


Figure 15: Generation of a PDDL domain file.

4.2. Automatic Generation of the Problem File

The PDDL problem file consists of dynamically generated `init` and `goal` states. The predicates and function initializations that go in the `goal` state are built from the OWL goal instance file. This information is then written in the `goal` state of the problem file. The `goal` state for a given kit to build is unchanged during replanning and plan repair. The `init` state allows successful replanning and plan repair processes where the current state of the world becomes the new `init` state of the problem file. The tool has the ability to read the SOAP OWL class and instance files along with the OWL `init` instance files to write only predicates that are true in the `init` state. Functions are also initialized in the `init` state. A mapping for each predicate and for the initialization of each function is programmed.

4.2.1. Building and Writing Predicates

Algorithm 1: Building a predicate

```
1  read OWLClass:PartsTray in OWL init instance file;
2  build partsTrayNodeSet = $[\text{partsTray}_i]_{i=1}^{i=n}$;
3  if partsTrayNodeSet is not empty then
4      for each $\text{partsTray}_i$ do
5          build partNodeSet = $[\text{part}_j : (\text{kit}_i, \text{hasPartsVessel\_Part})]_{i=1,j=1}^{i=n,j=o}$;
6          if partNodeSet is not empty then
7              for each $\text{part}_j$ do
8                  write $(\text{partsVessel-has-part partsTray}_i\ \text{part}_j)_{i=1,j=1}^{i=n,j=o}$
9              end
10         end
11     end
12 end
```

As mentioned earlier, a mapping is set for each predicate that is contained in the `predicates` section of a domain file. The mapping reads OWL files to fetch the information that is relevant for a given predicate. For instance, the predicate `partsVessel-has-part part_a_tray part_a_1` (line 26 in Figure 3) is true if the parts tray `part_a_tray` contains the part `part_a_1`. The algorithm developed to build and write this predicate in the init section of the problem file is described in Algorithm 1. The algorithm reads the class *PartsTray* from the OWL init instance file, retrieves all the individuals of this class (line 1), and inserts the individuals in a list of parts trays partsTrayNodeSet (line 2). For each partsTray_i in partsTrayNodeSet, the algorithm uses the object property hasPartsVessel_Part to retrieve all the parts (line 4). The domain of the object property hasPartsVessel_Part is the OWL class *PartsTray* and its range is the OWL class *Part*, that is, given a parts tray, hasPartsVessel_Part fetches all the parts contained in this parts tray. Each part_i is then stored in a list of parts partNodeSet (line 5). Finally, the algorithm writes the predicate `partsVessel-has-part` with proper OWL individuals as parameters, i.e., `part_a_tray` and `part_a_1` in this example.

Note that if the condition at line 3 is not satisfied, i.e., if there are no parts trays in the ontology, this predicate is not written in the init state of the problem file. Similarly, if partNodeSet is empty, i.e., there is no parts in the parts tray, the predicate is also not written.

4.2.2. Initializing Functions

All the functions defined in the domain file are initialized in the init section of the problem file. If a function has a reference parameter, the function is initialized for each instance of the reference parameter. For instance, the function `(quantity-of-parts-in-partstray ?partstray - PartsTray)` sets the number of parts in a parts tray. This function is initialized for each parts tray found in the OWL init instance file, i.e., for `part_a_tray` and `part_b_tray` as shown in Figure 3 at line 29 and at line 30, respectively.

The value that is used to initialize a function is either retrieved from the OWL init instance file or computed using data from the OWL init instance file. For instance, initializing the function `(capacity-of-kits-in-lbwk ?lbwk - LargeBoxWithKits)` (line 25 in Figure 2) requires a value which is retrieved, while the value that is used to initialize the function `(quantity-of-parts-in-partstray ?partstray - PartsTray)` (line 18 in Figure 2) is computed. The algorithm that builds and writes the former function is illustrated in Algorithm 2 and the algorithm that builds and writes the latter function is illustrated in Algorithm 3. Explanations for Algorithm 2 and Algorithm 3 are given as follows:

- Algorithm 2: First, the algorithm reads the OWL `init` instance file (line 1), retrieves all OWL individuals of type *LargeBoxWithKits*, and stores them in the list largeBoxWithKitsNodeSet (line 2). If the list largeBoxWithKitsNodeSet is not empty (line 3), for each $\mathrm{largeBoxWithKits_i}$, the number of kits that the large box with kits $\mathrm{largeBoxWithKits_i}$ can contain (lbwkCapacity) is retrieved with the OWL data property hasLargeWithKits_Capacity (line 5). Finally, the function is written with $\mathrm{largeBoxWithKits_i}$ as the reference parameter and lbwkCapacity as the initial value (line 6).

Algorithm 2: Retrieved value for function initialization.

```
1  read OWLClass:LargeBoxWithKits in OWL init instance file;
2  build largeBoxWithKitsNodeSet = $[\mathrm{largeBoxWithKits_i}]_{i=1}^{i=n}$;
3  if largeBoxWithKitsNodeSet is not empty then
4      for each $\mathrm{largeBoxWithKits_i}$ do
5          int lbwkCapacity = $[\mathrm{largeBoxWithKits_i}, \mathrm{hasLargeWithKits\_Capacity}]_{i=1}^{i=n}$;
6          write (= (capacity-of-kits-in-lbwk $\mathrm{largeBoxWithKits_i}$) lbwkCapacity)$_{i=1}^{i=n}$
7      end
8  end
```

- Algorithm 3: First, the algorithm reads the OWL `init` instance file (line 1), retrieves all OWL individuals of type *PartsTray*, and stores them in the list partsTrayNodeSet (line 2). For each parts tray $\mathrm{partsTray_i}$, the number of parts NumberOfParts in $\mathrm{partsTray_i}$ is set to 0 (line 5). The algorithm then retrieves all the parts for each $\mathrm{partsTray_i}$ and stores them in the list partNodeSet (line 6). NumberOfParts is incremented each time a part is found in partNodeSet (line 9). Finally, for each $\mathrm{partsTray_i}$, the function is initialized, where $\mathrm{partsTray_i}$ is the reference parameter and NumberOfParts is the number of parts in $\mathrm{partsTray_i}$ (line 12).

Algorithm 3: Computed value for function initialization.

```
1  read OWLClass:PartsTray in OWL init instance file;
2  build partsTrayNodeSet = $[\mathrm{partsTray_i}]_{i=1}^{i=n}$;
3  if partsTrayNodeSet is not empty then
4      for each $\mathrm{partsTray_i}$ do
5          set numberOfParts = 0;
6          build partNodeSet = $[\mathrm{part_j} : (\mathrm{partsTray_i}, \mathrm{hasPartsVessel\_Part})]_{i=1,j=1}^{i=n,j=o}$;
7          if partNodeSet is not empty then
8              for each $\mathrm{part_j}$ do
9                  numberOfParts + +
10             end
11         end
12         write (= (quantity-of-parts-in-partstray $\mathrm{partsTray_i}$) numberOfParts)$_{i=1}^{i=n}$
13     end
14 end
```

5. Conclusion and Future Work

This paper presented the latest tasks in the area of kit building, part of the "Agility Performance of Robotic Systems" (APRS) project at the National Institute of Standards and Technology (NIST). The authors presented models in the XML Schema Definition Language (XSDL) for PDDL structures. These models are used to generate a set of OWL files. The generated OWL files are employed to automatically and dynamically generate PDDL domain and problem files. This function allows replanning and plan repair when action failures are encountered during the kitting process. The automatic and dynamic generation of PDDL files is an attempt at bringing agility and flexibility in the APRS project. As mentioned in Section 2, a MySQL database is updated with information on the current state of the world as a kitting process is performed. The next step in this effort is to generate the PDDL problem file with data directly retrieved from the MySQL database while the PDDL domain file will still be generated from a set of OWL files.

6. Disclaimer

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.

References

[1] C. Schlenoff, Agility Performance of Robotic Systems, `http://www.nist.gov/el/isd/aprs.cfm`, accessed: 04-20-2014 (2013).

[2] G. Chryssolouris, Manufacturing Systems: Theory and Practice, second edition Edition, Mechanical Engineering, Springer, New York, NY, 2006.

[3] A. Gunasekaran, Agile Manufacturing: Enablers and an Implementation Framework, International Journal of Production Research 36 (5) (1998) 1223–1247.

[4] P. Lindbergh, Strategic Manufacturing Management: A Proactive Approach, International Journal of Operations and Production Management 10 (2) (1990) 94–106.

[5] H. Sharafi, Z. Zhang, A Method for Achieving Agility in Manufacturing Organisations: An Introduction, International Journal of Production Economics 62 (1–2) (1999) 7–22.

[6] K. Breu, C. Hemingway, M. Strathern, Workforce Agility: The New Employee Strategy for the Knowledge Economy, Journal of Information Technology 17 (2002) 21–31.

[7] K. Conboy, B. Fitzgerald, Toward a Conceptual Framework of Agile Methods: A Study of Agility in Different Disciplines, in: Proceedings of the 2004 ACM workshop on Interdisciplinary software engineering research (WISER), 2004, pp. 37–44.

[8] B. Tan, Agile Manufacturing and Management of Variability, International Transactions on Operational Research 5 (5) (1998) 375–388.

[9] R. D. Vor, J. Mills, Agile Manufacturing, American Society of Mechanical Engineers 2 (2) (1995) 977.

[10] A. Kusak, D. He, Design for Agile Assembly: An Operational Perspective, International Journal of Production Research 35 (1) (1997) 157–178.

[11] D. Upton, The Management of Manufacturing Flexibility, California Management Review 36 (2) (1994) 72–89.

[12] Y. Yusuf, M. Sarhadi, A. Gunasekaran, Agile Manufacturing: The Drivers, Concepts and Attributes, International Journal of Production Economics 62 (1) (1999) 23–32.

[13] M. Hong, S. Payander, W. Gruver, Modelling and Analysis of Flexible Fixturing Systems for Agile Manufacturing, in: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, 1996, pp. 1231–1236.

[14] M. Zain, N. Kassim, E. Mokhtar, Use of Information Technology and Information Systems for Organisational Agility in Malaysian Firms, Singapore Management Review 50 (1) (2003) 69–83.

[15] B. Kannan, L. Parker, Fault-Tolerance Based Metrics for Evaluating System Performance in Multi-Robot Teams, in: Proceedings of Performance Metrics for Intelligent Systems Workshop, 2006, pp. 54–61.

[16] P. J. P. Leitao, An Agile and Adaptive Holonic Architecture for Manufacturing Control, Ph.D. thesis, University of Porto (January 2004).

[17] M. Fox, A. Gerevini, D. Long, I. Serina, Plan Stability: Replanning versus Plan Repair, in: Proceedings of the International Conference on Automated Planning & Scheduling (ICAPS), The English Lake District, Cumbria, UK, 2006, pp. 212–221.

[18] M. Fox, D. Long, PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains, Journal of Artificial Intelligence Research 20 (2003) 431–433.

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

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

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

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

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

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

[25] J. Albus, 4-D/RCS Reference Model Architecture for Unmanned Ground Vehicles, in: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 2000, pp. 3260–3265.

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

[27] S. Balakirsky, Z. Kootbally, An Ontology Based Approach to Action Verification for Agile Manufacturing, in: Robot Intelligence Technology and Applications 2, Vol. 274 of Advances in Intelligent Systems and Computing, Springer, 2014, pp. 201–217.

[28] S. Balakirsky, T. Kramer, Z. Kootbally, A. Pietromartire, Metrics and Test Methods for Industrial Kit Building, NISTIR 7942, National Institute of Standards and Technology, Gaithersburg, MD (May 2013).

[29] ISO, 10303-11: 2004: Industrial automation systems and integration — Product data representation and exchange — Part 11 : Description method: The EXPRESS language reference manual.

[30] B. Stroustrup, C++ Programming Language, special Edition, Addison-Wesley, 2000.

[31] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL – The Planning Domain Definition Language, Tech. Rep. CVC TR-98-003/DCS TR-1165, AIPS-98 Planning Competition Committee, Yale Center for Computational Vision and Control (October 1998).

[32] A. E. Gerevini, A. Saetti, I. Serina, An Approach to Efficient Planning with Numerical Fluents and Multi-criteria Plan Quality, Artificial Intelligence 172 (8–9) (2008) 899–944.

[33] D. Mcdermott, The 1998 AI Planning Systems Competition, AI Magazine 21 (2) (2000) 35–55.

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

[35] The OWL API, `http://owlapi.sourceforge.net/`, accessed: 2014-02-21 (2014).