Available online at www.sciencedirect.com
ScienceDirect
Robotics and Computer-Integrated Manufacturing 00 (2014) 1–17
Robotics and Computer-Integrated Manufacturing

Models Development and Dynamic Generation of PDDL Files for Flexible Kit Building Applications

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

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

Abstract

This paper presents different models that are used to represent PDDL (Planning Domain Definition Language) structures for kit building applications. Moreover, a tool-based approach is described to automatically and dynamically generate PDDL files for replanning from the current state of the world to recover from failures. The effort described in this paper is an attempt at including agility and flexibility into the "Agility Performance of Robotic Systems" project, developed at the National Institute of Standards and Technology (NIST), and conducted in connection with the Working Group on Ontologies for Robotics and Automation of the IEEE Robotics and Automation Society.

Keywords: PDDL (Planning Domain Definition Language), OWL (Web Language Ontology), 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 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 they will be able to accurately predict the agility performance of their robot systems and be able to quickly re-task and reconfigure their assembly operations.

*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)

The increased number of new models and variants have forced manufacturing firms to meet the demands of a diversified customer base by producing 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] 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 Pinto-Leitão & Restivo [16] as the capability to remain working correctly and relatively stable, even in presence of disturbances.

The goal of the effort discussed in this paper is to provide a robust architecture for the "Agility Performance of Robotic Systems" project. More particularly, the current effort aims at planning and replanning for failure recovery (e.g., misalignments, incorrect parts and tooling, shortage of parts, or missing tool). Fox *et al.* [17] discussed plan replanning and plan repair when differences are detected between the expected and actual context of execution during plan execution in real environments. The 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.

The planning aspect of the APRS project 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. Both the domain and problem files are able to be auto-generated from a knowledge representation. 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. The kitting environment model and the kitting plan model have both been fully defined in each of two languages: XML Schema Definition Language (XSDL) [19, 20, 21] and Web Ontology Language (OWL) [22, 23, 24].

This paper is structured as follows: An overview of the knowledge driven methodology for the APRS project is presented in Section 2. The XML schemas models that were developed to represent PDDL domain and problem files are discussed in Section 3. A java-based tool that was developed to dynamically produce PDDL domain and problem files from the set of OWL files previously generated is described in Section 4. Finally, concluding remarks and future work are addressed in Section 5.

2. Knowledge Driven Methodology

The knowledge driven methodology presented in this section is not purposed 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.

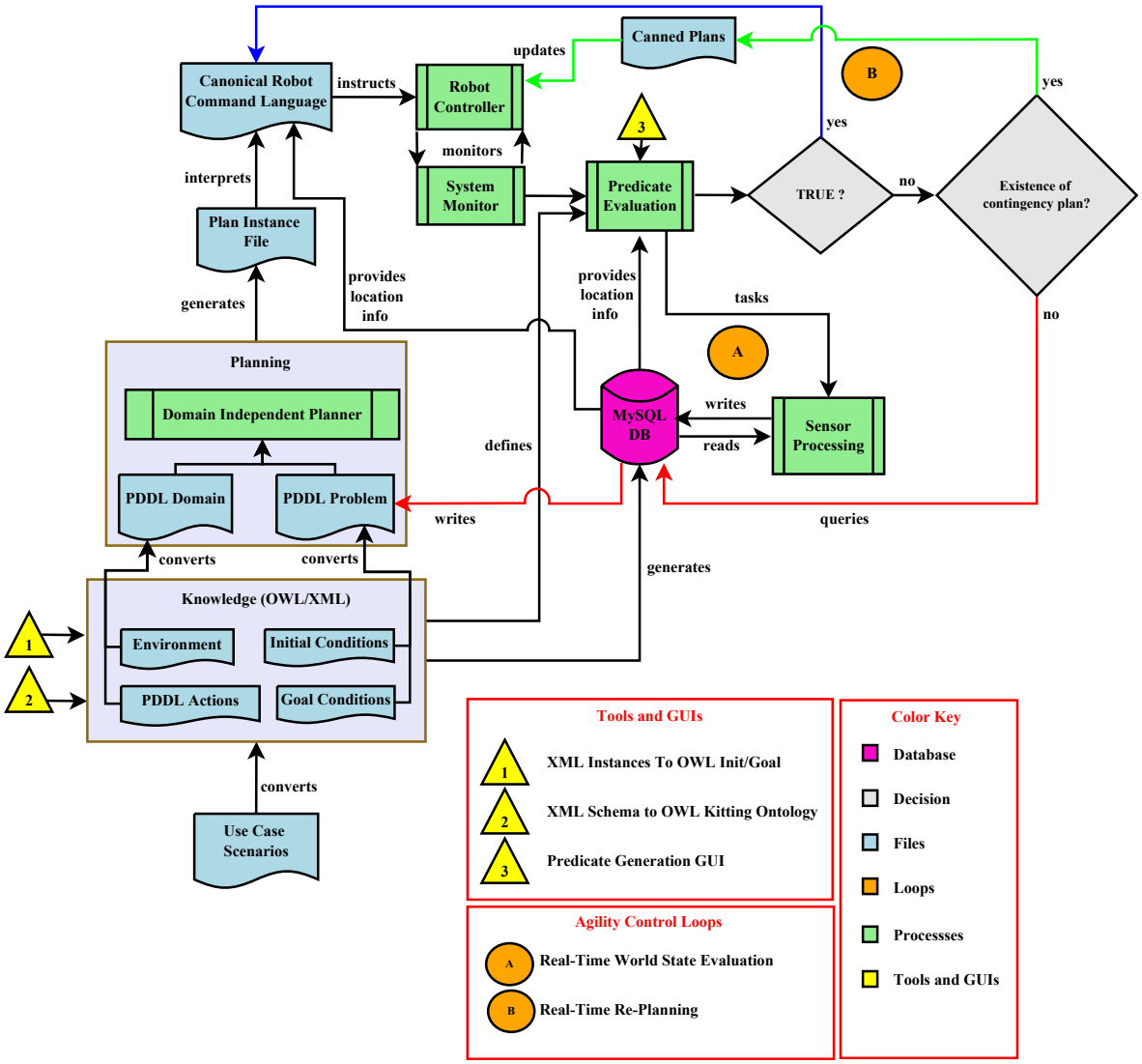

Figure 1: Knowledge Driven Design extensions.

- Use Case Scenarios – At the early stage of assembly, the Use Case Scenarios define the type of assembly that will be performed. When a new order comes from a customer, an operator enters the specifications of the order in the system via a graphical user interface (e.g., the type of assembly and the number of products required). This first step is therefore an attempt to introduce agility in our 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 and PDDL actions and of instance files that describe the initial and goal states for the current assembly. The knowledge is represented in a compact form with knowledge classes inheriting common attributes from parent classes. The PDDL Actions knowledge describes aspects of PDDL actions that are necessary 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 expressed with 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 PDDL Actions OWL files while the PDDL Problem file is produced from the Initial Condition and the Goal Condition files. From these two files, a domain independent planning system [26] was used to produce a static Plan Instance File.

- Canonical Robot Command Language – Once a plan has been formulated, the knowledge is transformed into a representation that is optimized for use by a robotic system (the Robot Language). The interpreter combines knowledge from the plan with knowledge from the MySQL database to form a sequence of sequential actions that the robot controller is able to execute. The authors devised a canonical robot command language (CRCL) in which such lists can be written. The purpose of the CRCL is to provide generic commands that implement the functionality of typical industrial robots without being specific either to the language of the planning system that makes a plan or to the language used by a robot controller that executes a plan.

- 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 where the oldest entry is processed first (FIFO).

- Predicate Evaluation – Prior to carrying out the current action, the System Monitor sends the action to a Predicate Evaluation process. The Predicate Evaluation process is used to check if the precondition section for the current PDDL action is true and is intrinsically used to identify failures during the execution of an action by the robot. Each precondition is a predicate expression that must be validated prior to action execution. The world model (the MySQL database) 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.

  Two distinct results come out of the Predicate Evaluation process. In one hand, all the predicates for the precondition section have been evaluated to true, in this case the robot is signaled to perform the current action. On the other hand, at least one predicate for the precondition section is evaluated to false, therefore a failure occurred in the system. 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 (Canned Plans), and the chance that this kind of failure could occur. When no failure modes exist for the failed predicate(s), replanning is performed. Before replanning takes place, the MySQL database is queried in order to build the initial state of the environment in the PDDL problem file. This way ensures that the initial state of the environment is properly set with current information that will be used to generate a new plan. A deeper analysis of the Predicate Evaluation process can be found in [27].

3. Models for PDDL

This section describes XML schema models that were developed to represent PDDL structures. In this project, a two-step process is required to generate PDDL files. First, XML schema and XML instance files are used to generate

a set of OWL files. Then, the generated OWL files are 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.

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 since 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 progress was 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 plan metrics is not currently used in our effort, it is the intention of the authors to do so as the project grows.

3.1.1. PDDL Domain File

Prior the development of a schema model, components of PDDL domain and problem files need to be identified. 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 different components of a PDDL domain file 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.
- lines 3–7: Parameter and object types have to be declared before they are used (before predicates and functions). This is done with the declaration (:types $name_1$ ... $name_n$).
- lines 9–20: 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.

```
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
9     (:predicates
10        (hasEndEffector-HeldObject-None ?endeffector - EndEffector)
11        (hasEndEffector-For-Sku-KitTray ?endeffector - EndEffector ?sku - StockKeepingUnit)
12        (hasRobot-EndEffector ?robot - Robot ?endeffector - EndEffector)
13        (hasEndEffector-PhysicalLocation-RefObject-Robot ?endeffector - EndEffector
14                                                         ?robot - Robot)
15        (hasKitTray-SkuObject-Sku ?kittray - KitTray ?sku - StockKeepingUnit)
16        (hasKitTray-PhysicalLocation-RefObject-LBWEKT ?kittray - KitTray
17                                                      ?lbwekt - LargeBoxWithEmptyKitTrays)
18        (hasEndEffector-HeldObject-KitTray ?endeffector - EndEffector ?kittray - KitTray)
19        (hasKitTray-PhysicalLocation-RefObject-EndEffector ?kittray - KitTray
20                                                           ?endeffector - EndEffector))
21
22    (:functions
23        (quantity-of-parts-in-partstray ?partstray - PartsTray)
24        (quantity-of-parts-in-kit ?sku - StockKeepingUnit ?kit - Kit)
25        (quantity-of-kittrays-in-lbwekt ?lbwekt - LargeBoxWithEmptyKitTrays)
26        (quantity-of-kits-in-lbwk ?lbwk - LargeBoxWithKits)
27        (current-quantity-kit ?kit - Kit)
28        (final-quantity-kit ?kit - Kit)
29        (capacity-of-parts-in-kit ?partsku - StockKeepingUnit ?kit - Kit)
30        (capacity-of-kits-in-lbwk ?lbwk - LargeBoxWithKits)
31        (part-found-flag))
32
33    (:action Take-KitTray
34        :parameters(
35            ?robot - Robot
36            ?kittray - KitTray
37            ?lbwekt - LargeBoxWithEmptyKitTrays
38            ?endeffector - EndEffector
39            ?sku - StockKeepingUnit)
40        :precondition(and
41            (> (quantity-of-kittrays-in-lbwekt ?lbwekt) 0)
42            (hasEndEffector-HeldObject-None ?endeffector)
43            (hasEndEffector-For-Sku-KitTray ?endeffector ?sku)
44            (hasRobot-EndEffector ?robot ?endeffector)
45            (hasEndEffector-PhysicalLocation-RefObject-Robot ?endeffector ?robot)
46            (hasKitTray-SkuObject-Sku ?kittray ?sku)
47            (hasKitTray-PhysicalLocation-RefObject-LBWEKT ?kittray ?lbwekt))
48        :effect(and
49            (decrease (quantity-of-kittrays-in-lbwekt ?lbwekt) 1)
50            (hasEndEffector-HeldObject-KitTray ?endeffector ?kittray)
51            (hasKitTray-PhysicalLocation-RefObject-EndEffector ?kittray ?endeffector)
52            (not (hasEndEffector-HeldObject-None ?endeffector))
53            (not (hasKitTray-PhysicalLocation-RefObject-LBWEKT ?kittray ?lbwekt))))
54 )
55
```

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

- lines 22–31: 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 33–53: The domain definition contains operators (called actions in PDDL). An action statement specifies a

way that a planner affects the state of the world. The statement includes parameters, preconditions, and effects. An action is identified by a unique name (Take-KitTray at line 33). Each parameter of an action consists of a name and a type (e.g., at line 35, robot is the name of the parameter and Robot is its type). Preconditions and effects may consist of positive predicates (lines 42–47 and lines 50–51), function operations (line 49), and conditions on numeric expressions (line 41). Conditions on numeric expressions are always comparisons between pairs of numeric expressions. Effects may also consist of negative predicates (lines 52–53).

3.1.2. PDDL Problem File

Figure 3 is a complete PDDL problem file created for kitting. The different components of a PDDL problem file are described below:

- 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: The keyword domain is a reference to the domain to which the problem is associated. In the example, the domain refers to the domain described in Figure 2.
- lines 3–17: objects specifies the distinct types of objects that will appear in the initial and goal states.
- lines 18–57: The init section consists of predicates that are true in the initial state. Because of the closed 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 also set in the init section. For instance, line 43 tells the planner that kit_1 has the capacity of holding only 1 part of type stock_keeping_unit_part_c. In Figure 3, function assignments are depicted at lines 40–57.
- lines 58–62: 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, that is, the SOAP model discussed in this section is used in multiple projects.

The types section in the domain and the objects section in the problem are stored in the KittingWorkstation model and this model is imported in 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].

Conceptually, the SOAP and KittingWorkstation models are object models as found in several object oriented programming languages (C++ , for example [30]). That is:

- the model consists primarily of class definitions,
- a class defines a type of thing,
- classes have attributes (”elements” in XML schema language),
- the class definition gives the class (or data type for individual variables) of each attribute,
- some attributes may occur optionally or multiple times,

```
(define (problem kitting-problem)
    (:domain kitting-domain)
    (:objects
        robot_1 - Robot
        changing_station_1 - EndEffectorChangingStation
        kit_tray_1 - KitTray
        kit_1 - Kit
        empty_kit_tray_supply - LargeBoxWithEmptyKitTrays
        finished_kit_receiver - LargeBoxWithKits
        work_table_1 - WorkTable
        part_a_tray part_b_tray - PartsTray
        part_a_1 part_a_2 part_b_1 - Part
        part_gripper tray_gripper - EndEffector
        part_gripper_holder tray_gripper_holder - EndEffectorHolder
        stock_keeping_unit_part_a stock_keeping_unit_part_b - StockKeepingUnit)
    (:init
        (hasPart-SkuObject-Sku part_a_1 stock_keeping_unit_part_a)
        (hasPart-SkuObject-Sku part_a_2 stock_keeping_unit_part_a)
        (hasPart-SkuObject-Sku part_b_1 stock_keeping_unit_part_b)
        (hasKit-KitTray kit_1 kit_tray_1)
        (hasKitTray-SkuObject-Sku kit_tray_1 stock_keeping_unit_kit_tray)
        (hasEndEffector-PhysicalLocation-RefObject-EndEffectorHolder part_gripper part_gripper_holder)
        (hasEndEffector-PhysicalLocation-RefObject-EndEffectorHolder tray_gripper tray_gripper_holder)
        (hasRobot-No-EndEffector robot_1)
        (hasWorkTable-ObjectOnTable-None work_table_1)
        (hasKitTray-PhysicalLocation-RefObject-LBWEKT kit_tray_1 empty_kit_tray_supply)
        (hasPart-PhysicalLocation-RefObject-PartsTray part_a_1 part_a_tray)
        (hasPart-PhysicalLocation-RefObject-PartsTray part_a_2 part_a_tray)
        (hasPart-PhysicalLocation-RefObject-PartsTray part_b_1 part_b_tray)
        (hasEndEffector-HeldObject-None part_gripper)
        (hasEndEffector-HeldObject-None tray_gripper)
        (hasPartsVessel-Part part_a_tray part_a_1)
        (hasPartsVessel-Part part_a_tray part_a_2)
        (hasPartsVessel-Part part_b_tray part_b_1)
        (hasEndEffector-For-Sku-KitTray tray_gripper stock_keeping_unit_kit_tray)
        (hasEndEffector-For-Sku-Part part_gripper stock_keeping_unit_part_a)
        (hasEndEffector-For-Sku-Part part_gripper stock_keeping_unit_part_b)
        (hasEndEffectorHolder-EndEffector part_gripper_holder part_gripper)
        (hasEndEffectorHolder-EndEffector tray_gripper_holder tray_gripper)
        (hasEndEffectorHolder-PhysicalLocation-RefObject-ChangingStation part_gripper_holder changing_station_1)
        (hasEndEffectorHolder-PhysicalLocation-RefObject-ChangingStation tray_gripper_holder changing_station_1)
        (hasEndEffectorChangingStation-EndEffectorHolder changing_station_1 part_gripper_holder)
        (hasEndEffectorChangingStation-EndEffectorHolder changing_station_1 tray_gripper_holder)
        (= (quantity-of-parts-in-partstray part_a_tray) 2)
        (= (quantity-of-parts-in-partstray part_b_tray) 1)
        (= (quantity-of-parts-in-kit stock_keeping_unit_part_a kit_1) 0)
        (= (quantity-of-parts-in-kit stock_keeping_unit_part_b kit_1) 0)
        (= (quantity-of-kittrays-in-lbwekt empty_kit_tray_supply) 1)
        (= (quantity-of-kits-in-lbwk finished_kit_receiver) 0)
        (= (capacity-of-kits-in-lbwk finished_kit_receiver) 12)
        (= (current-quantity-kit kit_1) 0)
        (= (final-quantity-kit kit_1) 3)
        (= (part-found-flag) 1))
    (:goal(and
        (= (capacity-of-parts-in-kit stock_keeping_unit_part_a kit_1) 2)
        (= (capacity-of-parts-in-kit stock_keeping_unit_part_b kit_1) 1)
        (hasKit-PhysicalLocation-RefObject-LBWK kit_1 finished_kit_receiver)
        (hasLBWK-Kit finished_kit_receiver kit_1)))
)
```

Figure 3: The PDDL problem file for kitting.

- some classes are derived from others; thus, there is a derivation hierarchy,

- a derived class has all the attributes of its parent plus, possibly, some of its own,
- if class B is derived from class A, then if the type of an attribute is class A, an instance of class B may be used as the value of the attribute,
- the model does not use multiple inheritance,
- the model also uses primitive data types such as numbers and strings, and provides for defining specialized data types by putting constraints on primitive data types.

To describe the SOAP model for PDDL domains, the authors will 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..∞ underneath a box means it may not occur, with no upper limit on the number of occurrences, and a 1..∞ underneath a box means it may occur at least once, with no upper limit on the number of occurrences.

Domain. DomainType extends DataThingType and consists of a Name (inherited from DataThingType), a set of requirements (Requirement), a set of variables (Variable), a set of predicates (PositivePredicateType), an optional set of functions (FunctionType), and a set of actions (ActionBaseType). These components consist a PDDL domain file such as the one showed in Figure 2. A Requirement and a Variable are of type `xs:NMTOKEN` in XSDL and of type `string` in OWL. The other components are described later in this section.

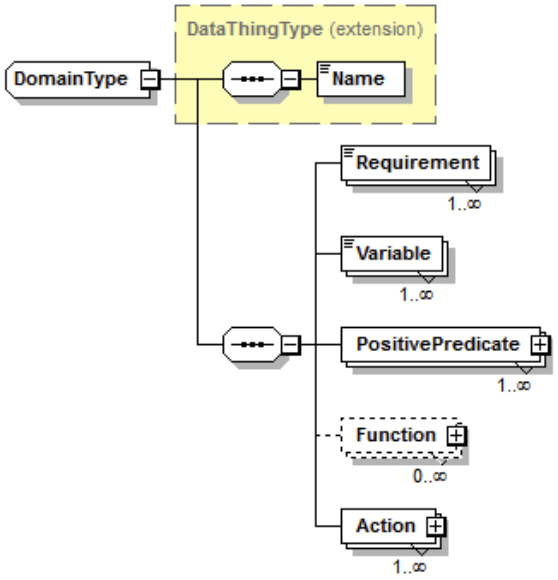

Figure 4: Domain model.

Predicate. PDDL predicates are represented in a positive and negative forms. Figure 2 contains both positive (e.g., line 42) and negative predicates (e.g., lines 10 and 53). A PDDL positive predicate is modeled with PositivePredicateType, which is depicted in Figure 5. A PositivePredicateType extends DataThingType and consists of a name (Name) that is inherited, an optional description (Description), a reference parameter (ReferenceParameter), and an optional target parameter (TargetParameter). 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 a reference parameter (ReferenceParameter) and the second parameter is identified as the target parameter (TargetParameter). In the case a predicate has only one parameter, this parameter is identified as the reference parameter (ReferenceParameter).

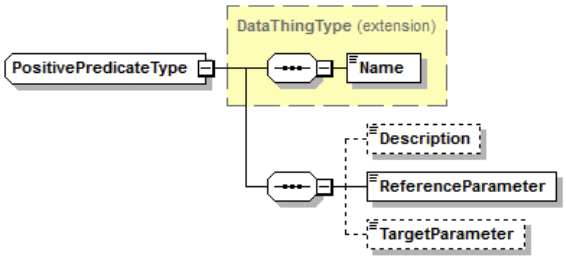

Figure 5: Positive predicate model.

Function. PDDL functions are represented by FunctionType. The structure of FunctionType is illustrated in Figure 6. The interpretation for ReferenceParameter and for TargetParameter are the same as the ones given earlier. As one can see, ReferenceParameter and TargetParameter are both optional for a FunctionType since some PDDL functions in our kidding domain are void of parameters (e.g., (part-found-flag) in Figure 2).

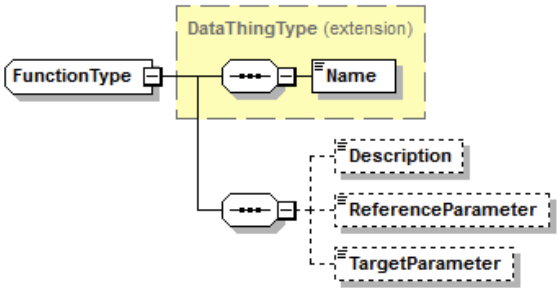

Figure 6: Function model.

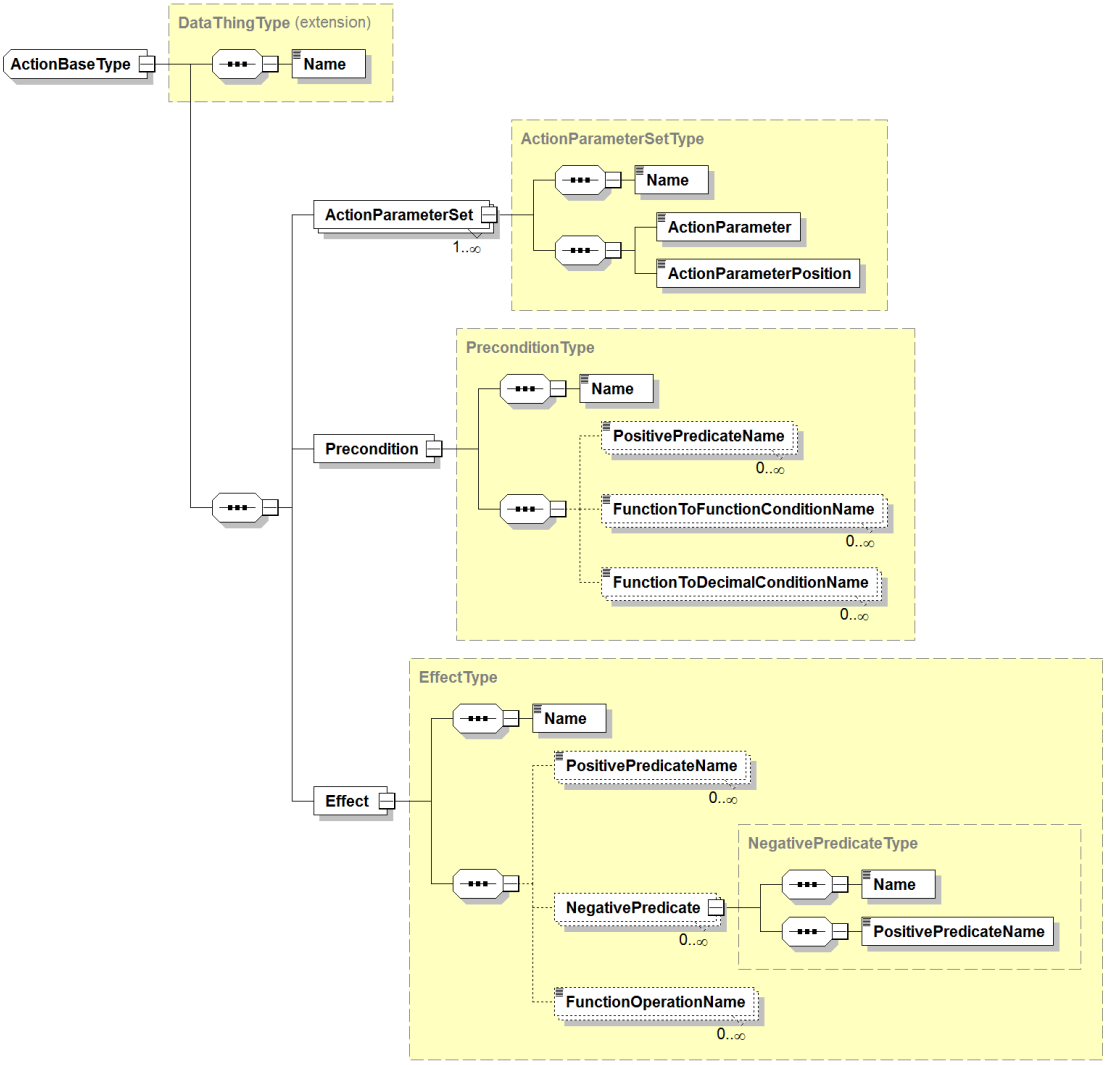

Figure 7: Action model.

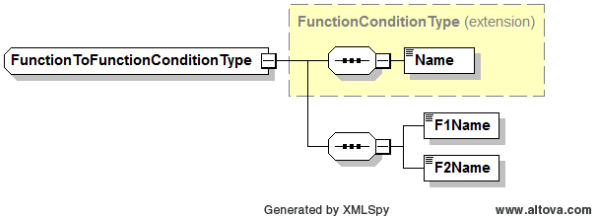

Figure 8: Function to function condition model.

Action. As mentioned in Section 3.1.1, a PDDL action consists of a unique name, a set of parameters, a precondition section, and an effect section. The model for PDDL actions is ActionBaseType (Figure 7). ActionBaseType extends DataThingType. The models for the components of a PDDL action are described as follows:

- The unique name of a PDDL action is assigned with the inherited Name attribute.

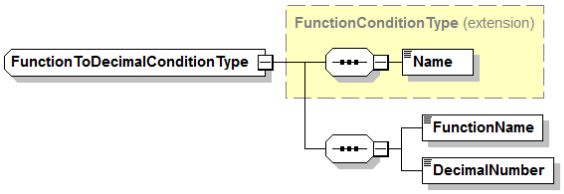

Figure 9: Function to decimal condition model.

- PDDL actions' parameters are represented with ActionParameterSetType. An ActionParameterSetType consists of a unique Name, ActionParameter is a text field (xs:NMTOKEN in XSDL) which represents the type of the parameter, and ActionParameterPosition is an integer (xs:positiveInteger in XSDL) which represents the position of the ActionParameter in the list of parameters for a PDDL action [1]. To illustrate these components, the reader may refer to Figure 2 (line 33). The action Take-KitTray consists of five parameters. Each parameter of the action Take-KitTray is an ActionParameterSetType. The ActionParameter for the parameter robot is Robot and its ActionParameterPosition is 1.

- The precondition section of a PDDL action is modeled with PreconditionType . A PreconditionType extends DataThingType. A PreconditionType consists of a unique Name (inherited), optional references to positive predicates (PositivePredicateType), optional references to conditions on functions (FunctionConditionType). A FunctionConditionType is a comparison between pairs of numeric expressions. It includes a function to function condition model and a function to decimal condition model.

  - A function to function condition model (FunctionToFunctionConditionType) is represented in Figure 8. The model extends FunctionConditionType and consists of a Name (inherited from DataThing), a reference to the first FunctionType (*F1Name*), and a reference to the second FunctionType (*F2Name*). Comparisons between FunctionTypes require definitions of mathematical symbols. Specific mathematical symbols ($<$, $\leq$, $=$, $\geq$, $>$) are expressed with subtypes of FunctionConditionType. The subtypes are FunctionToFunctionLessType (Figure 10), FunctionToFunctionLessOrEqualType (Figure 11), FunctionToFunctionEqualType (Figure 12), FunctionToFunctionGreaterOrEqualType (Figure 13), and FunctionToFunctionGreaterType (Figure 14).
  - A function to decimal condition model (FunctionToDecimalConditionType) is represented in Figure 9. The model extends FunctionConditionType and consists of a Name (inherited from DataThing), a reference to a FunctionType (*FunctionName*), and a decimal value (*DecimalNumber*). As for FunctionToFunctionConditionType, subtypes of FunctionToDecimalConditionType indicates the type of the comparison between a function and a decimal number. The subtypes of FunctionToDecimalConditionType are FunctionToDecimalLessType (Figure 15), FunctionToDecimalLessOrEqualType (Figure 16), FunctionToDecimalEqualType (Figure 17), FunctionToDecimalGreaterOrEqualType (Figure 18), and FunctionToDecimalGreaterType (Figure 19).

    An illustration of a FunctionToDecimalGreaterType is given at line 41 in Figure 2. (> (quantity-of-kittrays-in-lbwekt ?lbwekt) 0) involves the mathematical symbol > to compare the function (quantity-of-kittrays-in-lbwekt ?lbwekt) with a decimal.

- The effect section of a PDDL action is modeled with EffectType. An EffectType extends DataThingType. An EffectType consists of a unique Name (inherited from DataThing), optional references to PositivePredicateTypes (*PositivePredicateName*), optional NegativePredicateTypes, and optional references to FunctionOperationTypes (*FunctionOperationName*).

[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 list in OWL.

– A NegativePredicateType models the negation of a predicate (e.g., (not (hasEndEffector-HeldObject-None ?endeffector))). The model for negative predicates is depicted in Figure 20. A negative predicate extends DataThing and consists of a Name (inherited from DataThing) and a reference to a PositivePredicateType (*PositivePredicateName*). Note that the model uses a reference and not the definition of the positive predicate. This requires the existence of the positive predicate before it is used by the negative predicate.
– A FunctionOperationType models arithmetic expressions on functions. Effects can make use of a selection of assignment operations in order to update the values of primitive numeric expressions. An instance of a function operation, (decrease (quantity-of-kittrays-in-lbwekt ?lbwekt) 1), is given at line 49 in Figure 2. A FunctionOperationType extends DataThingType. A FunctionOperationType consists of a unique Name (inherited from DataThing), a reference to a Functiontype (referred with *FunctionName*), an Expression, and a Value. In the function operation given above, the FunctionType is (quantity-of-kittrays-in-lbwekt ?lbwekt), the Expression is decrease, and the Value is 1.

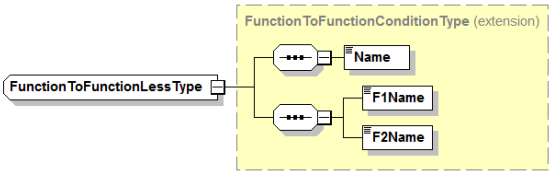

Figure 10: Function to function less model.

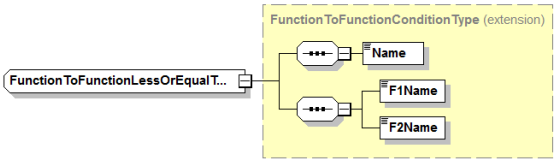

Figure 11: Function to function less or equal model.

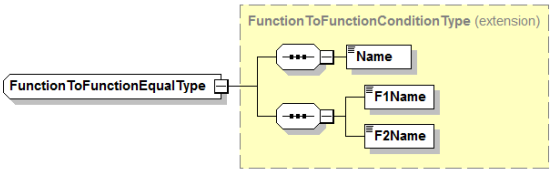

Figure 12: Function to function equal model.

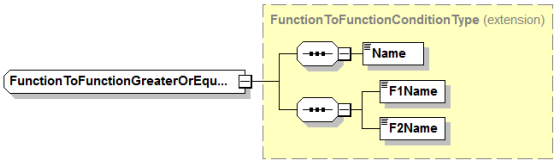

Figure 13: Function to function greater or equal model.

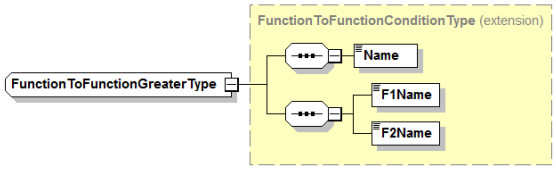

Figure 14: Function to function greater model.

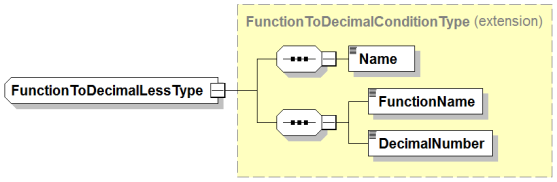

Figure 15: Function to decimal less model.

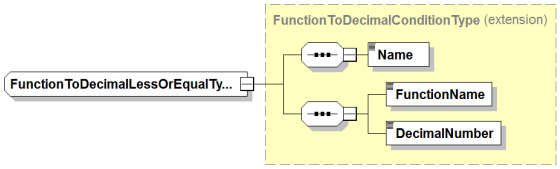

Figure 16: Function to decimal less or equal model.

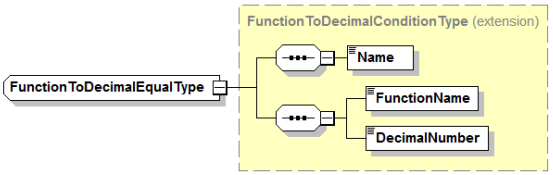

Figure 17: Function to decimal equal model.

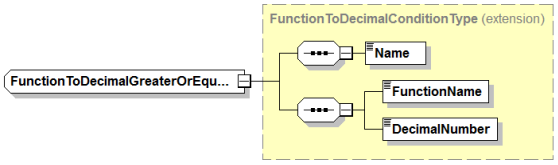

Figure 18: Function to decimal greater or equal model.

4. Automatic Generation of PDDL Files

Once the schema models for PDDL 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 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

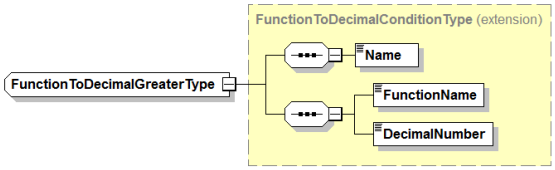

Figure 19: Function to decimal greater model.

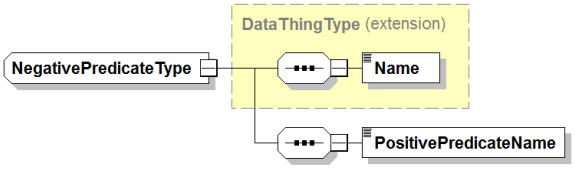

Figure 20: Negative predicate model.

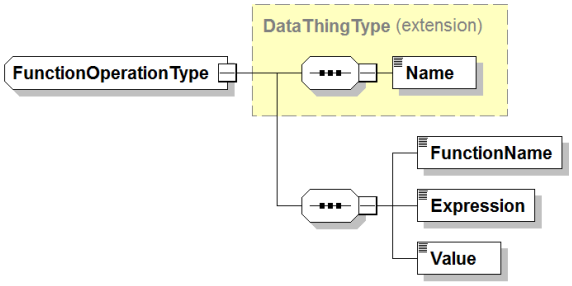

Figure 21: Function operation model.

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 presented 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.

The generation of the PDDL domain file is performed by reading the OWL classes from the SOAP OWL file that define PDDL structures. 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. The PDDL problem file, however, consists of a dynamically generated init state. This dynamic generation of the init state allows a successful replanning process 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 file and to write only predicates and function operations that satisfy the init state. In other words, a mapping for predicates and function operations is programmed and the tool writes predicates and function operations only if relevant information exists for those predicates and function operations.

For instance, the predicate hasPartsVessel-Part part_a_tray part_a_1 (line 32 in Figure 3) is true if the parts tray part_a_tray contains the part part_a_1. The process to build and write this predicate in the init section of the problem file is described in Figure 1. The different prerequisites for this predicate to be written in the init state of the problem file are described as follows. The tool reads the class PartsTray from an OWL init file, retrieves all the individuals in this class (line 1) and inserts them in $partsTrayNodeSet$ (line 2). If $partsTrayNodeSet$ is not empty (line 3),

Algorithm 1: Dynamic building and writing of a predicate

```
1  read OWLClass:PartsTray in OWL file;
2  build partsTrayNodeSet = [partsTray_i]_{i=1}^{i=n};
3  if partsTrayNodeSet is not empty then
4      for each partsTray_i do
5          build partNodeSet = [part_j : (kit_i, hasPartsVessel_Part)]_{i=1,j=1}^{i=n,j=o};
6          if partNodeSet is not empty then
7              for each part_j do
8                  write (hasPartsVessel-Part partsTray_i part_j)_{i=1,j=1}^{i=n,j=o}
9              end
10         end
11     end
12 end
```

for each *partsTray* (line 4) the tool uses the object property *hasPartsVessel_Part* to retrieve all the parts. The parts are then stored in *partNodeSet* (line 5). If *partNodeSet* is not empty (line 6), then the tool writes the predicate partsVessel-has-part with the correct name for the individuals (part_a_tray and part_a_1 in our example). Note that if the condition at line 3 is not satisfied, i.e., if there is no parts trays in the ontology, this predicate is not written in the initial 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 in the problem file.

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" project at the National Institute of Standards and Technology (NIST). The authors presented different models 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 a system to cope with failures with the production of new problem files based on the current state of the world. The automatic and dynamic generation of PDDL files is an attempt at bringing agility and flexibility in the current kitting system. As mentioned in Section 2, a MySQL database is used to store information on the current state of the world. The next step in this effort is to generate PDDL files directly from the MySQL database which contains the latest snapshot of the world.

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. P. L. ao, 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).