

International Journal of Semantic Computing
© World Scientific Publishing Company

IMPLEMENTATION OF AN ONTOLOGY-BASED APPROACH TO ENABLE AGILITY IN KIT BUILDING APPLICATIONS

ZEID KOOTBALLY

*Department of Aerospace and Mechanical Engineering, University of Southern California
Los Angeles, California 90089, USA
zeid.kootbally@nist.gov*

THOMAS R. KRAMER

*Department of Mechanical Engineering, Catholic University of America
Washington, DC 20064, USA
thomas.kramer@nist.gov*

CRAIG SCHLENOFF

*Intelligent Systems Division, National Institute of Standards and Technology
Gaithersburg, Maryland 20899, USA
craig.schlenoff@nist.gov*

SATYANDRA K. GUPTA

*Department of Aerospace and Mechanical Engineering, University of Southern California
Los Angeles, California 90089, USA
skgupta@usc.edu*

Received (Day Month Year)

Revised (Day Month Year)

Accepted (Day Month Year)

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 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 focuses on the information models and describes how they are used to introduce robot agility within the kitting domain. 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. Details are given on the automatic generation of OWL class model files from XML schema model files. Files of OWL instances conforming to an OWL class model are generated automatically from XML instance files by automatically-built translators. Furthermore, a design methodology for kit building is presented and shows how the different information

2 *kootbally, kramer, schlenoff, gupta*

models are used with the other components of the methodology.

Keywords: Web Ontology Language; kit building; robotics; agility; knowledge representation; information model.

1. Introduction

Robots will be a pervasive part of our lives in the coming decades. Whether it is behind the scenes in assembling products that we use in our everyday lives, or helping us parallel park a car, robots are already playing a role in what we use and how we get around.

However robots are not good at everything. For the most part, robots perform best in highly structured environments, where objects are in well-known, predictable locations. Robots are also not known for “thinking on the fly” very well. They are best when they can be trained to perform a very specific activity, which requires a very specific set of motions, and that activity can be performed in the exact same way many hundreds or thousands of times. Not surprisingly, robots have been adopted much more in high volume, repeatable operations such as car manufacturing than they have been in smaller job shop type operations where only a handful of similar products are being made at a given time.

Another way to describe this is that robots are not considered agile. But, in order for them to be useful to small manufacturers and to also allow larger manufacturers to offer more automated customization of high volume parts (think cars and cell phones), they need to be. The Agility Performance of Robotic Systems (APRS) project at the National Institute of Standards and Technology (NIST)^a is addressing this challenge. The goal of this project is to enable agility in manufacturing robot systems and to develop the measurement science which will allow manufacturers to assess and assure the agility performance of their robot systems. Key areas of robot agility include: 1) the ability of a robot to be rapidly re-tasked without the need to shut down the robot for an extended period of time when a new operation needs to be performed, 2) the ability of a robot to recover from errors, so that when a part is dropped, for example, the robot can assess the situation and determine the best way to proceed to accomplish the goal, and 3) the ability to quickly swap in and out robots from different manufacturers so that a company is not tied to a single robot brand.

NIST is developing software, standards, and performance metrics to allow aspects of agility to happen. NIST is developing the Canonical Robot Command Language (CRCL) [1] which is a low-level messaging language for sending commands to, and receiving status from, a robot. CRCL is intended primarily to provide commands that are independent of the kinematics of the robot that executes the commands. This allows robots to be more easily swapped in and out since the robot commands are represented in a robot-agnostic format. NIST is also in the

^a<https://www.nist.gov/programs-projects/agilityperformance-robotic-systems>

process of developing a set of performance metrics to measure robot agility, and is validating them at an upcoming Agile Robotics for Industrial Automation Competition (ARIAC)^b to be held in 2017 in conjunction with the Institute of Electrical and Electronics Engineers (IEEE).

One of the key aspects needed to enable robot agility is the ability for the robot to represent knowledge about the environment (and its own capabilities) in such a way that it can reason over it and take action based on what it has learned. NIST has chaired an IEEE Working Group which developed the IEEE 1872 Standard (Core Ontology for Robotics and Automation (CORA)) [2]. This standard defines a core ontology that allows for the representation of, reasoning about, and communication of knowledge in the robotics and automation (R&A) domain. This ontology includes generic concepts as well as their definitions, attributes, constraints, and relationships. These terms can be specialized to capture the detailed semantics for concepts in robotics sub-domains. The standard has chosen to use the Standard Upper Ontology Knowledge Interchange Format (SUO-KIF) [3] to represent concepts and their associated axioms.

In this paper, the authors describe how the concepts in three ontologies in the Web Ontology Language (OWL) [4] format, consistent with CORA, were specialized to enable agility in industrial robotics and the infrastructure that was built to convert the concepts into various representations that could be directly applied to the robot control system. The effort described in this paper deals with kitting or kit building. In kitting, parts are delivered to the assembly station in kits that contain the exact parts necessary for the completion of one assembly object.

This paper is structured as follows: Section 2 describes the information models used within this project. Section 3 describes the tools used to translate XML (eXtensible Markup Language) Schema Definition Language (XSDL) and XML files to OWL files. Section 4 provides details on XML to OWL translation. Section 5 describes the design methodology that relies on OWL files to perform kitting in the APRS project. Section 6 provides an overview of the implementation of agile robotics at NIST. Section 7 gives conclusions and future work.

2. Information Models

The APRS project makes use of three ontologies that can be applied to the kitting domain. The knowledge is represented in as compact of a form as possible with knowledge classes inheriting common attributes from parent classes. The authors used a set of closely related C++ software tools for manipulating XSDL [5, 6, 7] files and XML instance files and translating them into OWL class files and OWL instance files [8]. The authors used the translators to translate XML instance files to OWL instance files. The OWL class files are used as input to a tool that generates a database schema automatically. The APRS work in OWL generation was reported

^b<https://www.nist.gov/el/intelligent-systems-division-73500/agile-robotics-industrial-automation>

4 *kootbally, kramer, schlenoff, gupta*

in [8]. Modest improvements have been implemented since then.

In the diagrams presented in the following subsections, a dotted line around a box means the attribute is optional (may occur zero times), while a $..∞$ underneath a box means it may occur more than once, with no upper limit on the number of occurrences. A + sign at the right of a box means that the item in the box has substructure not currently shown. If there is no + sign, the item is primitive data such as a number or a string of characters.

2.1. *Kitting Workstation Model*

The KittingWorkstation model (Figure 1) describes the objects in the current kitting scenario. This model contains generic information and classes that are needed for the domain of kit building. The ontology contains information on basic elements such as a “point” (Figure 2a) which is defined as a class that contains a name and a three-dimensional quantity, as well as complex types such as a “part”, which is shown in Figure 2b, and contains elements such as the part’s location and a name that references a stock keeping unit. The stock keeping unit (SKU) (Figure 3) contains static information on classes of parts such as the part’s shape, weight, and the end effector that should be used for grasping the part. Both static and dynamic information is represented in this ontology and is automatically transitioned into the planning and execution areas of the World Model.

Another area of the ontology contains specific instances needed for a particular kitting domain. For example, it contains the definition of the finished kits that may be constructed and specific information on the individual parts. A finished kit is an instance of the PartsTrayType class (Figure 4). It contains information on parts and their locations within the kit, information on its completeness which is a Boolean value set to true if the kit has all the required parts in their designed slot. A slot is a non-physical object that contains information on its relative location within a kit, the part that it contains (optional), and a Boolean value set to true if the slot is occupied by a part.

One of the goals of the APRS project is to introduce additional agility into the kit building process. Therefore, partial information is accepted and even encouraged for this area of the ontology. For the example of a part shown in Figure 2b, information on the SKU, grasp points (part of the ExternalShape or InternalShape), and name would be expected to be available at runtime. Information on the location of the part (PrimaryLocation) may not become valid until after a sensor processing system has identified and located the particular part. More information on this model can be found in [9, 10].

2.2. *Action Model*

The second ontology in the APRS project contains the high-level concept of an action and all of the concepts that are required to support an action. In this case, a Planning Domain Definition Language (PDDL) [11] action is being represented.

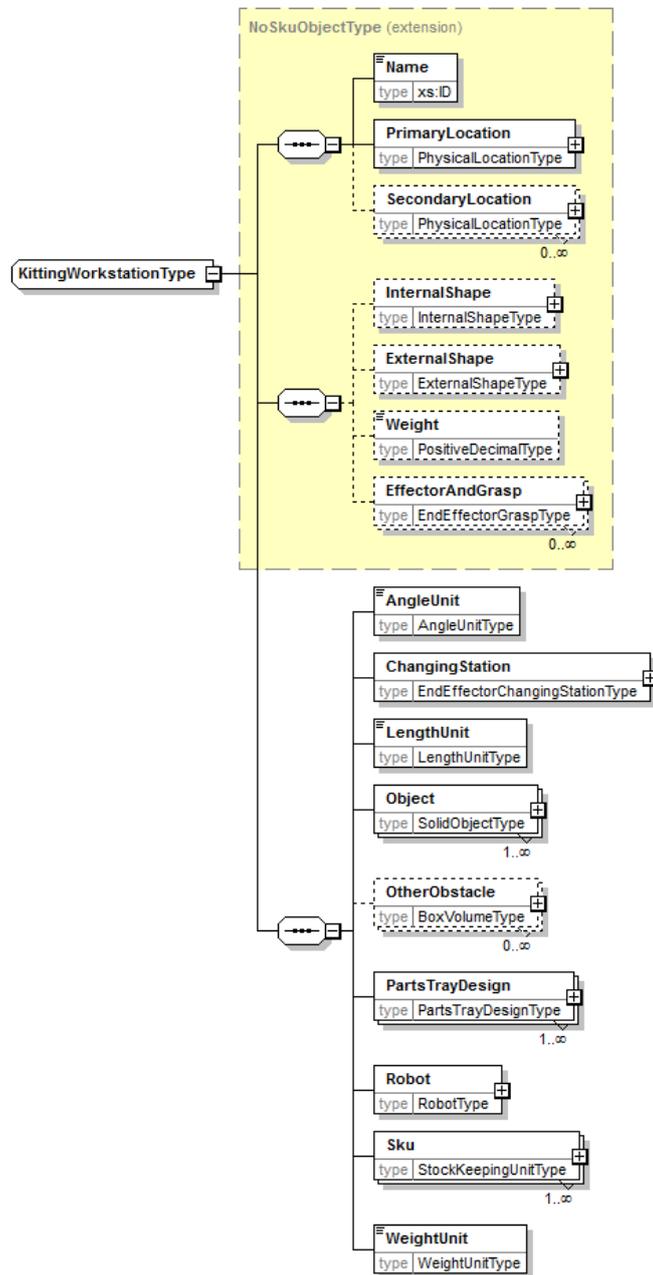


Fig. 1: Information model for kitting.

6 *kootbally, kramer, schlenoff, gupta*

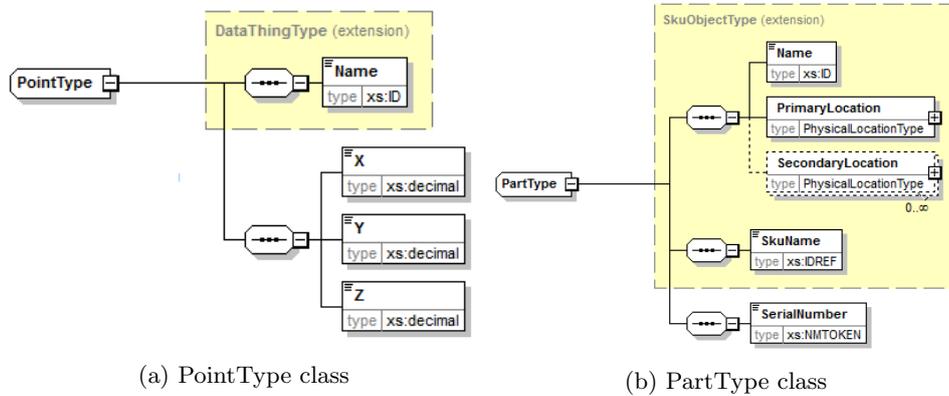


Fig. 2: Description of the PointType class that is designed to contain dynamic information and the PartType class that is designed to contain both static and dynamic information about particular parts.

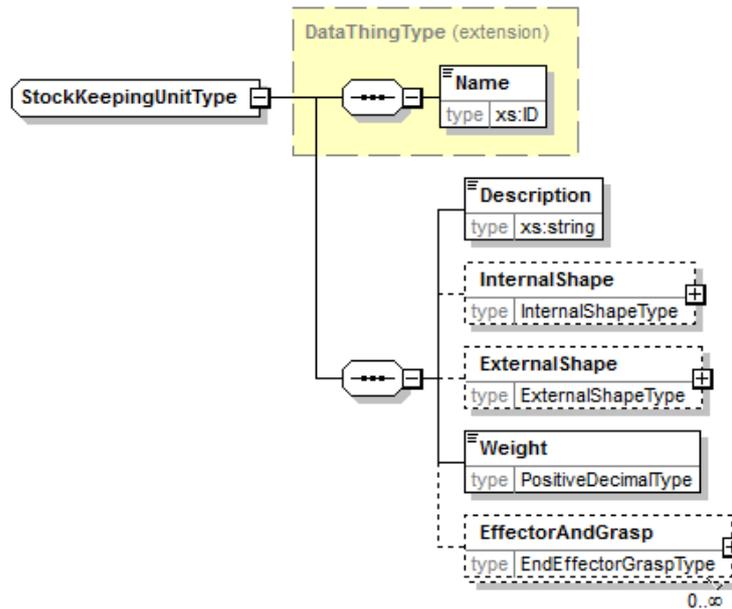


Fig. 3: Description of the StockKeepingUnitType class that contains static information about classes of parts.

PDDL is a community standard for the representation and exchange of planning domain models. 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

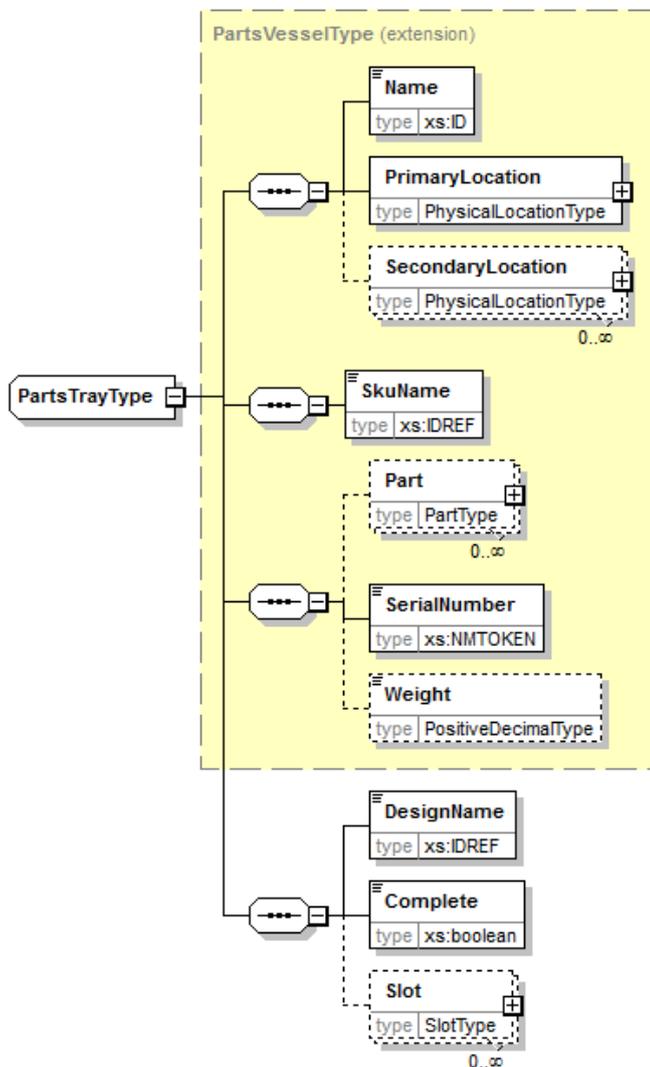


Fig. 4: Description of the `PartsTrayType` class that contains information on parts and their locations with a parts tray.

planning system creates an additional static plan file. The authors explored the idea of automating the generation of PDDL domain and problem files by representing the components of a PDDL domain file in an Action model [12]. A PDDL action is defined as an operator that causes one or more properties of an instance to change. Before this action may be performed, certain preconditions must be satisfied, and after the action is performed, certain effects will take place. The action accepts parameters that specify the particular instances that will be affected, where an

8 *kootbally, kramer, schlenoff, gupta*

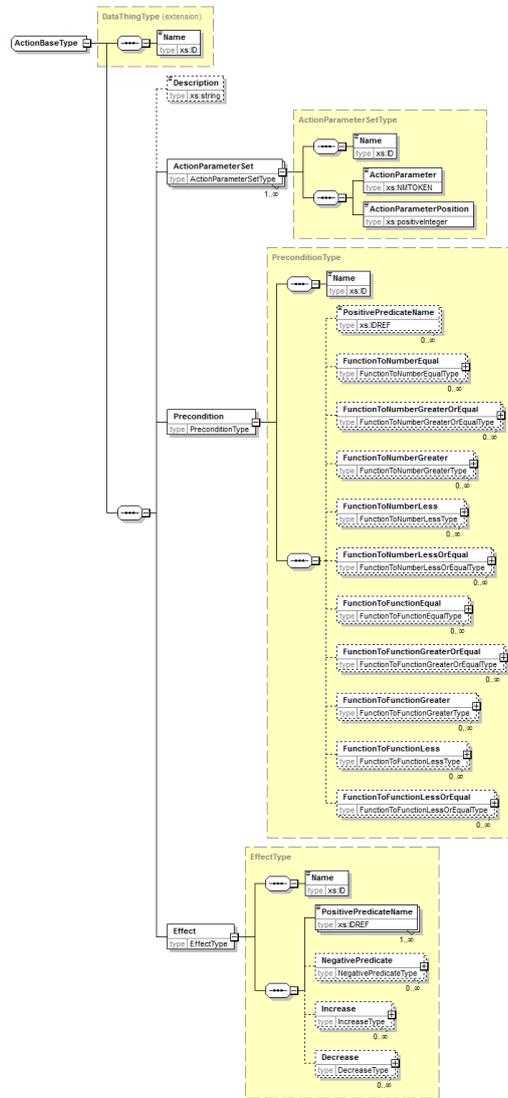


Fig. 5: Information model for PDDL actions.

instance refers to a physical object or piece of grounded data in the world. All of the necessary information for the automatic generation of the PDDL domain file required by the planning system is contained in this section of the ontology. The classes used to represent the actions in the ontology are illustrated in Figure 5. A more detailed analysis for this ontology is provided in [12].

2.3. Robot Capability Model

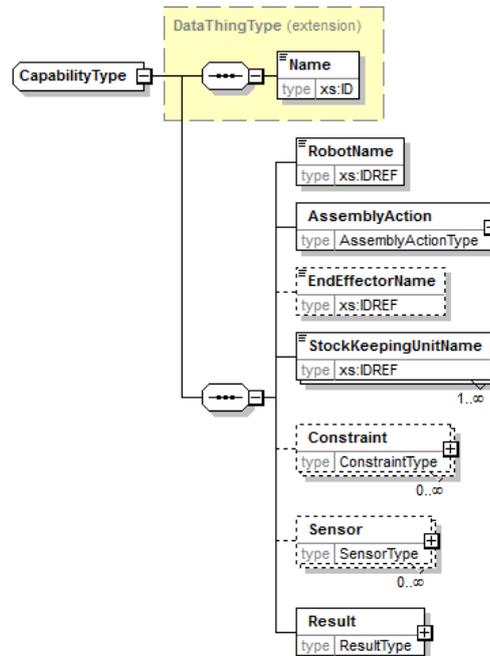


Fig. 6: Information model for robot capability.

The application of robotics in manufacturing assembly is hindered by their lack of agility, their large changeover times for new tasks and new products, and their limited reusability. One of the main causes for these hindrances is the lack of understanding of robot capabilities as they pertain to assembly tasks. In this context, a robot capability refers to the ability of a robot to perform a specific action on a specific object or set of objects. Measuring robot capabilities for a specific domain provides multiple advantages such as creating a process plan that assigns the best robot to each operation needed to accomplish the given task.

The Robot Capability model depicted in Figure 6 was produced based on the set of components identified from a thorough literature review [13]. The capability model consists of pointers to elements defined in the KittingWorkstation model as well as new elements. Mandatory elements include a pointer to the robot element for which the capability is defined (e.g., *robot-motoman*), the definition for the assembly action performed by the robot (e.g., *pick-up-part*), one or multiple references to the stock keeping unit for the object involved in the assembly action (e.g., *small-gear*), and the overall result for the current capability (e.g., can pick up this part 85 % of the time).

3. Overview of OWL Generation

A number of systems for converting XML files to OWL files have been developed. A survey of nine of these systems was made by Hacherouf *et al.* [14]. That survey does not include the system developed by the authors in the NIST Intelligent Systems Division (ISD), in the paper called the ISD OWL Generation System [8].

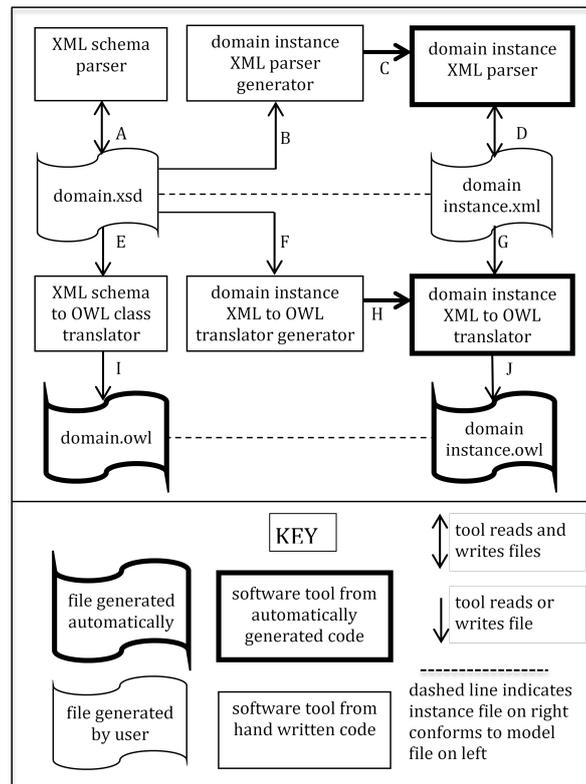


Fig. 7: ISD OWL Generation System.

The ISD system is built in C++ , using YACC and Lex [15] for parsing. C++ is generated from YACC by bison [16] and generated from Lex by flex [17]. As shown in Figure 7, the ISD system includes much more than XML to OWL translation. With reference to the lettered arrows on the figure, the capabilities of the system are:

- An XML schema model, domain.xsd, is built or imported. There is no representation of building it on Figure 7.
- Optionally, the XML Schema Parser is used to check that domain.xsd is valid (Arrow A). The XML Schema Parser reads domain.xsd, checking

schema validity while reading, and builds an abstract syntax tree representing domain.xsd in terms of objects conforming to a C++ class model of XSDL. The tree is then printed into the file domain.xsdecho, which may be compared to domain.xsd or ignored. The underlying parser used in the XML Schema Parser is used also in steps E and F, described below; it may also be compiled into other programs.

- The domain.xsd file is processed by the Domain Instance XML Parser Generator (Arrow B) to generate C++ , YACC, and Lex code for parsing XML instance files that conform to domain.xsd (Arrow C). A Domain Instance XML Parser is compiled from the code. The generated code includes a C++ class model of domain.xsd.
- An XML instance file, domainInstance.xml is built or imported. There is no representation of building it on Figure 7. This may be any XML instance file conforming to the domain.xsd schema.
- Optionally, the Domain Instance XML Parser is used to check that domainInstance.xml conforms to domain.xsd (Arrow D). The parser builds an abstract syntax tree conforming to the class model of domain.xsd. The tree is then printed into the file domainInstance.xml1, which may be compared to domainInstance.xml or ignored. The underlying parser used in the Domain Instance XML Parser is used also in step G, described below; it may also be compiled into other programs.
- The domain.xsd file is processed by the XML Schema to OWL Class Translator to produce the domain.owl file, which is the OWL class model equivalent to domain.xsd (Arrows E and I).
- Optionally, an OWL tool such as Protégé is used to check that domain.owl is valid (not shown on the figure).
- The domain.xsd file is processed by the Domain Instance XML to OWL Translator Generator to generate code for a translator that translates XML instance files conforming to domain.xsd into OWL instance files conforming to domain.owl. The Domain Instance XML to OWL Translator is compiled from the code (Arrows F and H). The parser used in the translator is the one produced by the Domain Instance XML Parser Generator that is used also in the Domain Instance XML Parser.
- The domainInstance.xml file is processed by the Domain Instance XML to OWL Translator to make the domainInstance.owl file, which conforms to domain.owl (Arrows G and J).
- Optionally, an OWL tool is used to check that domainInstance.owl is a valid instance file with respect to domain.owl.

OWL reasoners can make use of logical semantics, but cannot make any use of the real-world semantics (i.e., the meaning) of the model given in the in-line documentation. The authors run a reasoner in Protégé every time they check an OWL file in order to be sure the model is correct and they have not omitted something

that should be modeled.

Details of OWL generation (the bottom half of Figure 7) are discussed in the following section of this paper. Various improvements have been made to the ISD system since publication of [8]. The purely XML parts were updated most recently in May and October, 2016. The OWL parts were updated in October 2016 to work with the May version. Where this document differs from [8], this document is a correct description of the current version.

The ISD OWL Generation System differs from other XML to OWL systems in three significant ways. First, it uses Lex and YACC for parsing so that parsing is faster than in other systems. Second, for instance files, the system generates code for a translator. The compiled translator may then be used to translate any number of XML instance files into OWL instance files. Third, when instance files are translated, references from one object to another using `xs:ID` and `xs:IDREF` are resolved.

A minor difference from other systems is that all files written by the ISD system are carefully formatted for reading by humans; this is irrelevant to parsers but helpful for developers.

4. Translation Details

XML schema files and instance files are translated into OWL files having functional style syntax [18]. The mapping of XSDL constructs to OWL constructs implemented in the XML Schema to OWL Class Translator is shown in Table 1. Most of the mappings are the same as those in other XSDL to OWL class generators. Major exceptions are described in the next three paragraphs.

XML worlds are closed. In order to have the mapped OWL world duplicate the semantics of the XML world from which it is mapped despite OWL's open world assumption, for derived types, DisjointUnion declarations are made in the OWL class file. These declarations say that each parent type is the disjoint union of all of its derived types.

In order that certain database capabilities would exist in the database schema derived automatically from the OWL class file, it was decided to have explicit inverse object properties. This is indicated by the *hadByY_X* property shown in the table.

In order that there would be a useful name for every OWL NamedIndividual, every `complexType` in the schema file being translated must have a Name element of type ID. The Name element is not translated into the OWL class file. However, when an instance file is translated, the value of the Name element is used as the name of a NamedIndividual.

The most commonly used XSDL constructs not translated to an OWL class file are shown in Table 2. Several other less commonly used constructs not shown in the table are also not translated. Those that can be handled by converting them to a different construct in the schema are noted in Table 2. All the suggested conversions are trivial to make. The authors have written most of the schemas themselves

Table 1: XSDL to OWL mapping

XSDL Construct	OWL Construct
built-in datatype	built-in XML datatype
simpleType enumeration	DatatypeDefinition/DataOneOf
simpleType number restriction	max/min Inclusive/Exclusive
complexType	class declaration
derived type	SubClassOf declaration and DisjointUnion declaration
element named Name (of type ID)	deleted
element Y in complexType X	ObjectProperty $hasX_Y$ ObjectProperty $hadByY_X$ ObjectPropertyDomain (2) ObjectPropertyRange (2)
0 or 1 element (by default or by minOccurs and/or maxOccurs)	FunctionalDataProperty $hasX_Y$
element other than Name of built-in type or simpleType	DataProperty DataPropertyDomain DataPropertyRange
1 or more elements (by default, minOccurs and/or maxOccurs)	EquivalentClasses using $hasX_Y$
1 complexType element (default, minOccurs and/or maxOccurs)	InverseFunctionalObjectProperty
annotation/documentation	AnnotationAssertion
key	HasKey (needs improvement)
include	import
sequence	implicitly mapped by elements order is lost (usually irrelevant)
owlPrefix in documentation node	namespace with prefix

that have been used with OWL conversion, so they have not been hampered by unmapped constructs. If other constructs were essential to the work, the authors would have implemented them.

The mapping of XML instance files to OWL instance files is shown in Table 3. In addition to the explicitly mapped items shown in the table, the Prefix statements required at the beginning of the OWL file are inserted automatically by the automatically generated translator.

Table 2: XSDL constructs not mapped to OWL

XSDL Construct	Comment
attribute	convert attribute to element in schema
anonymous type	convert unnamed to named type in schema
choice	use xsi:type in instance instead if possible
keyref constraint	
max/minLength	
min/maxOccurs limits	e.g., maxOccurs="5"
namespace	but OWL namespaces are used
pattern	
ref	convert ref to element in type in schema
simpleType List	
substitutionGroup	use xsi:type in instance instead
unique constraint	

Table 3: XMLINSTANCE to OWL mapping

XML Instance Construct	OWL Construct
Name element (an xs:ID)	name of NamedIndividual
element instance	NamedIndividual Declaration
type of complexType element	ClassAssertion
complexType value	ObjectPropertyAssertion
built-in or simple value	DataPropertyAssertion
nesting level in hierarchy	comments giving nesting level
SchemaLocation of class file	import
named objects of same type	DifferentIndividuals

5. Design Methodology

The authors have implemented the design methodology shown in Figure 8 to perform kitting with an industrial robot.

A manufacturing activity begins when the operator receives a request to fulfill an order. The request is submitted to a Task Scheduler that starts the kitting process. To ensure real-time access to information in the environment, the authors developed a Java tool (Database Generator) that generates graph databases from the three ontologies described earlier. A graph representation conveys the properties of an ontology in a simple, clear, and structured model. To generate the graph databases, the authors followed the formal definition of an ontology given by Necib *et al.* [19]. The latter authors defined an ontology as a set of concepts ζ and a set of relationships \mathfrak{R} , where $c_i \in \zeta$ is a concept name, and $r_i \in \mathfrak{R}$ is a binary

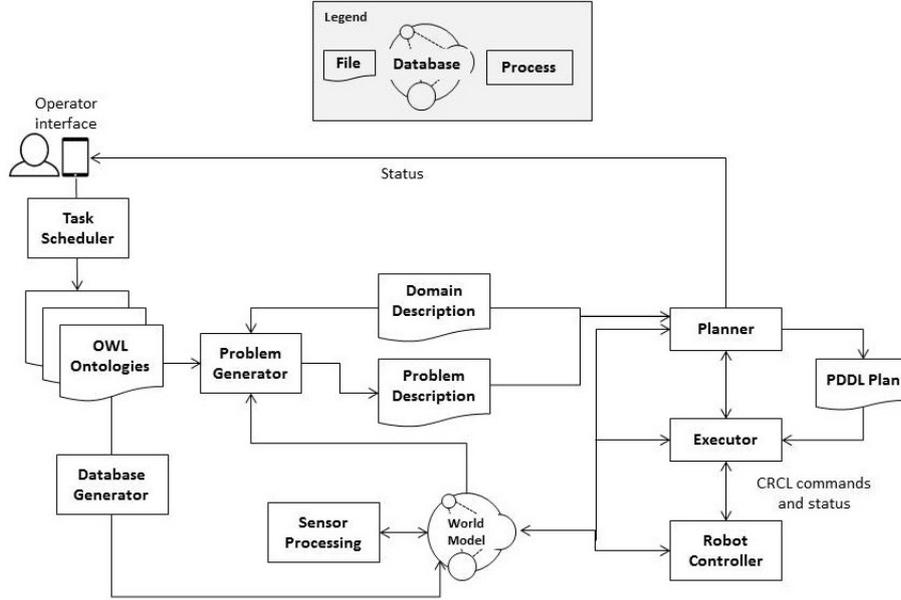


Fig. 8: Design methodology for kitting.

relationship between two concepts. c_i and r_i are non-null strings. An ontology is represented as a directed graph $G(V, E)$ where V is a finite set of vertices and E is a finite set of edges. Each vertex of V is labeled with a concept and each edge of E represents the relation between two concepts. Formally, the label of a node $n \in V$ is defined by a function $N(n) = c_i \in \zeta$ that maps n to a string from ζ . The label of an edge $e \in E$ is given by a function $T(e)$ that maps e to a string from \mathfrak{R} . Finally, an ontology is given by the set $O = \{G(V, E), \zeta, \mathfrak{R}, N, T\}$. Figure 9 gives an example of a graph representation in Neo4j^c of a selected portion from the ontology KittingWorkstation.

Using the Action and Robot Capability ontologies, the Problem Generator dynamically creates a PDDL problem file for the current domain. The Problem Generator parses the PDDL domain file to retrieve all the predicates and numeric-valued fluents (including robot capability values) and then uses a mapping file to retrieve their values in the graph database.

The current state of the world is updated by the Sensor Processing (SP) since objects in the environment could have been inadvertently moved since the previous time the database was updated. Once the problem file is auto-generated, both the problem and the domain files are used with a temporal planner to generate a plan file.

Next, an Executor application translates each PDDL action from the plan file

^c<https://neo4j.com/>

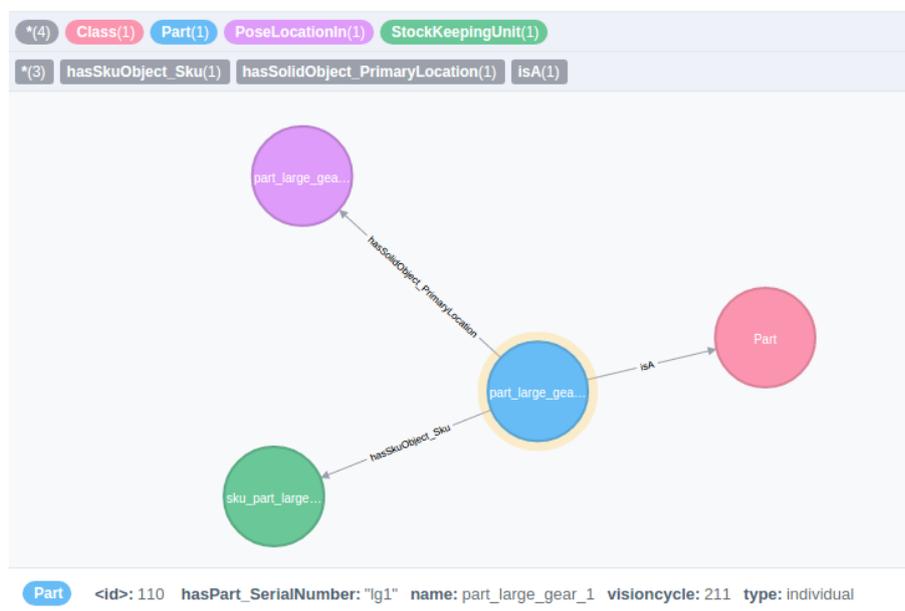


Fig. 9: Graph representation for the concept Part in the Neo4j graph database.

into a series of Canonical Robot Command Language (CRCL) commands, and sends each to a robot controller. CRCL provides message content and syntax for a low-level robot command-status protocol. The Executor application continually tracks the execution status of each CRCL command, ensuring that the conditions and effects of the PDDL commands are met as the statements are processed. If failures are detected, e.g., dropped part, the Executor aborts the PDDL plan and calls for replanning. SP continually estimates the state of the environment and synthesizes higher-level information about the state of the world by fusing lower-level sensor information.

6. Implementation

In order to determine an appropriate and relevant implementation of the methodology described in Section 5, the team at NIST did the following:

- Reviewed numerous robotics roadmaps, including “A Roadmap for US Robotics: From Internet to Robotics” [20].
- Conducted telecons and performed site visits to over 20 industrial and academic partners.
- Held discussions at various standards meetings and conferences, such as the International Conference on Robotics and Automation (ICRA), the Intelligent Robots and Systems (IROS) Conference, and the International

Conference on Automation Science and Engineering (CASE).

From this research and interaction, the following themes relating to robot agility became evident:

- Robots take a long time to program.
- Robots are incapable of adapting to changing environments.
- Once a company decides on a robot brand, they are tied to that brand because of the large infrastructural cost.
- Training a robot to perform a new task (or a variation of an existing task) is time consuming, tedious, and not cost effective unless the plant is dealing with very large lot sizes.
- Companies have large areas of their shop floor sitting idle because the robots were trained to develop a specific product and the demand for that product is low (even though demand for other products is high).

From this information, the authors developed two scenarios that are meant to represent starting points to address some of these challenges. Both scenarios involve kitting operations with two robots of different brands. The scenarios are described below:

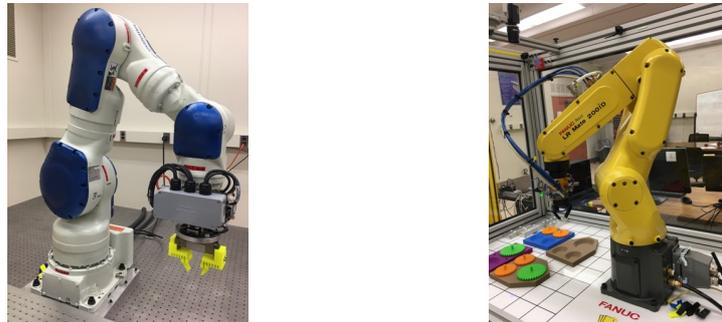


Fig. 10: Robots used at NIST to perform kitting.



Fig. 11: Kit models used during kitting scenarios at NIST.

- **Robot Failure and Retasking:** Two robots (Figure 10) are able to be quickly programmed using the methodology described in Section 5 to build two different kits (Figure 11). During the operation, the robot that is performing the higher priority kitting operation fails and is removed from the workcell. The second robot, from a different manufacturer, is able to be quickly re-tasked to complete the kit. No new code is written and the downtime is minimal. After the second robot completes the first robot's task, it goes back to its original kitting operation.

The most significant challenge in this scenario deals with the fundamentally different types of underlying representations (programming languages) that are used by the different robots. The Fanuc robot uses the Karel programming language and the Motoman robot uses the Inform programming language.

- **Task Failure Identification and Recovery:** During a kitting operation, a robot fails to properly perform a pick and place operation. The robot detects the failure, replans to determine how to fix the error, and corrects the error on its own. A failure in this context consists of incomplete kits, i.e., a finished kit that is missing one or more parts. To identify failures, information from the vision system and the database is used to deduce the completeness or incompleteness of finished kits. In the case of kits' incompleteness, high-level replanning is performed and robot actions are generated to bring the kits to a state of completeness.

7. Conclusions and Future Work

The Agility Performance of Robotic Systems project aims at making industrial robots more agile to tackle the challenges that small and medium manufacturers are facing. Implementing agility within the APRS project is performed within the kitting domain and starts with the definition of three information models which are consistent with the CORA model. This paper describes a KittingWorkstation model, an Action model, and a Robot Capability model. The KittingWorkstation model describes a kitting workstation including solid objects (e.g., parts) and data (e.g., poses). The Action model describes the structure components of a PDDL domain file and is used to automate the generation of PDDL domain and problem files. The Robot Capability model expresses the capability of a robot to perform specific actions on a specific object or set of objects. Although significant efforts have been made to improve agility in manufacturing kitting, there is still a need to deal with action failures. It is the intention of the authors to develop a new model for failures. This model will encompass information on the different failures that a robot can encounter during kitting as well as the severity of the failures and a remediation plan for each type of failure.

Disclaimer

Certain commercial hardware, open source software, and tools are identified in this paper in order to explain our research. Such identification does not imply recommendation or endorsement by the authors or the National Institute of Standards and Technology (NIST), nor does it imply that the software tools identified are necessarily the best available for the purpose.

Acknowledgements

Dr. Kramer acknowledges support for this work under grant 70NANB15H053 from the National Institute of Standards and Technology to the Catholic University of America.

Dr. Kootbally acknowledges support for this work under grant 70NANB15H249 from the National Institute of Standards and Technology to the University of Southern California.

References

- [1] F. Proctor, S. Balakirsky, Z. Kootbally, T. Kramer, C. Schlenoff, and W. Shackelford, “The Canonical Robot Command Language (CRCL),” *Industrial Robot: An International Journal*, vol. 43, no. 5, pp. 495–502, 2016.
- [2] S. R. Fiorini, J. L. Carbonera, P. Gonçalves, V. A. Jorge, V. F. Rey, T. Haidegger, M. Abel, S. A. Redfield, S. Balakirsky, V. Ragavan, H. Li, C. Schlenoff, and E. Prestes, “Extensions to the Core Ontology for Robotics and Automation,” *Robotics and Computer-Integrated Manufacturing*, vol. 33, pp. 3–11, Jun 2015, special Issue on Knowledge Driven Robotics and Manufacturing.
- [3] A. Pease, “Standard Upper Ontology Knowledge Interchange Format,” 2009. [Online]. Available: <http://suo.ieee.org/suokif.html>
- [4] World Wide Web Consortium, “OWL 2 Web Ontology Language Structural Specification and Functional Syntax,” 2009. [Online]. Available: <http://www.w3.org/TR/owl2-syntax/>
- [5] —, “XML Schema Part 0: Primer, Second Edition,” 2004. [Online]. Available: www.w3.org/TR/xmlschema-0
- [6] —, “XML Schema Part 1: Structures, Second Edition,” 2004. [Online]. Available: www.w3.org/TR/xmlschema-1
- [7] —, “XML Schema Part 2: Datatypes, Second Edition,” 2004. [Online]. Available: www.w3.org/TR/xmlschema-2
- [8] T. R. Kramer, B. H. Marks, C. I. Schlenoff, S. B. Balakirsky, Z. Kootbally, and A. Pietromartire, “Software Tools for XML to OWL Translation,” National Institute of Standards and Technology, Gaithersburg, MD, USA, NIST IR 8068, Jul 2015.
- [9] S. Balakirsky, Z. Kootbally, C. Schlenoff, T. Kramer, and S. Gupta, “An Industrial Robotic Knowledge Representation for Kit Building Applications,” in *Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vilamoura, Portugal: IEEE, October 2012, pp. 1365–1370.
- [10] S. Balakirsky, Z. Kootbally, T. Kramer, R. Madhavan, C. Schlenoff, and M. Shneier, “Functional Requirements of a Model for Kitting Plans,” in *Proceedings of the Workshop on Performance Metrics for Intelligent Systems*. ACM, 2012, pp. 29–36.

20 *kootbally, kramer, schlenoff, gupta*

- [11] M. Fox and D. Long, "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 431–433, 2003.
- [12] Z. Kootbally, C. Schlenoff, C. Lawler, T. Kramer, and S. Gupta, "Towards Robust Assembly with Knowledge Representation for the Planning Domain Definition Language (PDDL)," *Robotics and Computer-Integrated Manufacturing*, vol. 33, pp. 42–55, 2015, special Issue on Knowledge Driven Robotics and Manufacturing.
- [13] Z. Kootbally, "Industrial Robot Capability Models for Agile Manufacturing," *Industrial Robot: An International Journal*, vol. 43, no. 5, pp. 481–494, 2016.
- [14] M. Hacherouf, S. Bahloul, and C. Cruz, "Transforming XML Documents to OWL ontologies: A survey," *Journal of Information Science*, vol. 41, no. 2, pp. 242–259, 2015.
- [15] D. Brown, J. Levine, and T. Mason, *Lex & Yacc*. OReilly Media, October 1992.
- [16] C. Donnelly and R. Stallman, "Bison, The YACC-compatible Parser Generator," 2006. [Online]. Available: <http://dinosaur.compilertools.net/bison>
- [17] V. Paxson, W. Estes, and J. Millaway, "Flex, Version 2.5.31 A Fast Scanner Generator," 2003. [Online]. Available: <http://www.gnu.org/software/flex>
- [18] World Wide Web Consortium, "OWL 2 Web Ontology Language Structural Specification and Functional Style Syntax (Second Edition) W3C Recommendation 11 December 2012," 2012. [Online]. Available: <http://www.w3.org/TR/owl2-syntax>
- [19] C. B. Necib and J.-C. Freytag, *Ontology Based Query Processing in Database Management Systems*. Springer Berlin Heidelberg, 2003, pp. 839–857.
- [20] J. Baird, G. Bradski, M. Branicky, R. Brooks, J. Burke, H. I. Christensen, J. Enright, C. Flannigan, P. Freeman, T. Fuhlbrigge, J. Gemma, S. Gupta, G. Hildebrand, V. Kumar, P. Luh, M. Mason, E. Messina, E. Nieves, C. Richardson, D. Rus, M. Spong, J. Tsai, and S. Shepherd, "Roadmap for Robotics in Manufacturing," in *A Roadmap for U.S. Robotics – From Internet to Robotics*, 2013, pp. 7–25.