NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

Intelligent Systems Division

Knowledge Driven Planning and Modeling for Part Handling

Planning for a Kitting Workstation

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

Contents

1 Introduction

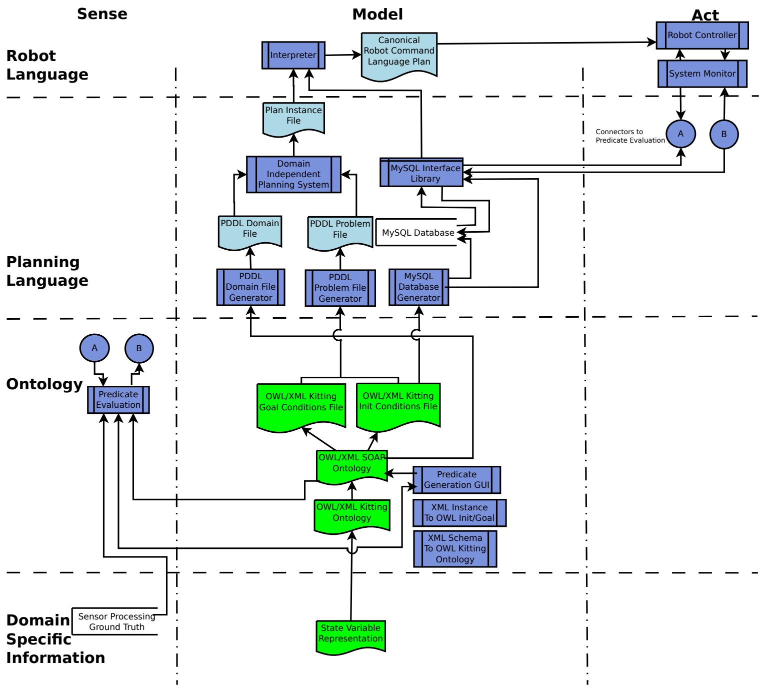

Sense
Model
Act
Robot Language
Planning Language
Ontology
Domain Specific Information
Interpreter
Canonical Robot Command Language Plan
Robot Controller
System Monitor
Plan Instance File
Connectors to Predicate Evaluation
A
B
Domain Independent Planning System
MySQL Interface Library
PDDL Domain File
PDDL Problem File
MySQL Database
PDDL Domain File Generator
PDDL Problem File Generator
MySQL Database Generator
Predicate Evaluation
OWL/XML Kitting Goal Conditions File
OWL/XML Kitting Init Conditions File
OWL/XML SOAP Ontology
Predicate Generation GUI
OWL/XML Kitting Ontology
XML Instance To OWL Init/Goal
XML Schema To OWL Kitting Ontology
Sensor Processing Ground Truth
State Variable Representation

Figure 1: Knowledge Driven Design extensions – In this figure, green shaded boxes with curved bottoms represent hand generated files while light blue shaded boxes with curved bottoms represent automatically created boxes. Rectangular boxes represent processes and libraries.

The knowledge driven methodology presented in this section is not purposed to act as a standalone system architecture. Rather it is intended to be an extension to well developed hierarchical, deliberative architectures such as 4D/RCS [1]. The overall knowledge driven methodology of the system is depicted in Figure 1. The figure is organized vertically by the representation that is used for the knowledge and horizontally by the classical sense-model-act paradigm of intelligent systems. The remainder of this section gives a brief description of each level of the hierarchy to help the reader understand the basic concepts implemented within the system architecture. The reader may find a more detailed description of each component and each level of the architecture in other documented publications [2, 3].

- On the vertical axis, knowledge begins with Domain Specific Information (DSI) that captures operational knowledge that is necessary to be successful in the particular domain in which the system is designed to operate. This includes information on items ranging from what actions and attributes are relevant, to what the necessary conditions are for an action to occur and what the likely results of the action are. The authors have chosen to encode this basic information in a formalism known as a state variable representation [7].

- At the next level up, the information encoded in the DSI is then organized into a domain independent representation. A base ontology (OWL/XML Kitting) contains all of the basic information that was determined to be needed during the evaluation of the use cases and scenarios. The knowledge is represented in as compact a form as possible with knowledge classes inheriting common attributes from parent classes. The OWL/XML SOAP ontology describes not only aspects of actions and predicates but also the individual actions and predicates that are necessary for the domain under study. The instance files describe the initial and goal states for the system through the Kitting Init Conditions File and the Kitting 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.

- 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). The planning language used in the knowledge driven system is expressed with the Planning Domain Definition Language (PDDL) [6] (version 3.0). 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 the ontology. The domain file represents actions along with required preconditions and expected results. The problem file represents the initial state of the system and the desired goal. From these two files, a domain independent planning system [4] was used to produce a static Plan Instance File.

- 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.

This document describes each level of the architecture from the Domain Specific Information level to the Robot Language level. The goal of this document is to help a user understand the different tools and techniques to generate the appropriate files in order to build a kit.

Each of the following section corresponds to each level of the architecture described in Figure 1 starting from the bottom level to the top level of the architecture.

2 Domain Specific Information

The foundation for the knowledge representation is domain specific information that is produced by an expert in the particular field of study. This includes information on items ranging from what actions and attributes are relevant, to what the necessary conditions are for an action to occur and what the likely results of the action are. We have chosen to encode this basic information in a formalism know as a state variable representation (SVR) [7]. This information will then flow up the abstraction and be transformed into the ontology, planning language, and robot language.
Before building a SVR, the domain for kitting needs to be specified. The domain for kitting contains some fixed equipment: a robot, a work table, end effectors, end effector holders, and an end effector changing station. Items that enter the workstation include kit trays, boxes in which to put kits, boxes that contain empty kit trays, and part supplies. Items that leave the workstation may be boxes with finished kits inside, empty part trays, empty boxes. An external agent is responsible of moving the items that leave the workstation. We assume that the workstation has only one work table, one changing station, and one robot.

In a State Variable Representation (SVR), each state is represented by a tuple of values of n state variables $\{x_1, \dots, x_n\}$, and each action is represented by a partial function that maps this tuple into some other tuple of values of the n state variables.

To build the SVR, the group has taken a very systematic approach of identifying and modeling the concepts. Because the industrial robot field is so broad, the group decided to limit its efforts to a single type of operation, namely kitting. A scenario was developed that described, in detail, the types of operations that would be performed in kitting, the sequencing of steps, the parts and machines that were needed, constraints on the process such as pre- and post-conditions, etc. For this scenario, a set of concepts were extracted and defined. These concepts served as the initial requirements for the kitting SVR. The concepts were then modeling in our SVR, building off of the definitions and relationships that were identified in the scenario. A SVR relies on the elements of constant variable symbols, object variable symbols, state variable symbols, and planning operators. These are defined for the kitting domain in the rest of this section.

2.1 Constant Variable Symbols

For the kitting domain, there is a finite set of constant variable symbols that must be represented. In the SVR, constant variable symbols are partitioned into disjoint classes corresponding to the objects of the domain. The finite set of all constant variable symbols in the kitting domain is partitioned into the following sets:

- A set of *Part*: A *Part* is the basic item that will be used to fill a kit.
- A set of *PartsTray*: *Parts* arrive at the workstation in *PartsTrays*. Each *Part* is at a known position in the *PartsTray*. Each *PartsTray* contains one type of *Part*.
- A set of *KitTray*: A *KitTray* can hold *Parts* in known positions.
- A set of *Kit*: A *Kit* consists of a *KitTray* and, possibly, some *Parts*. A *Kit* is empty when it does not contain any *Part* and finished when it contains all the *Parts* that constitute a kit.

- A set of *WorkTable*: A *WorkTable* is an area in the kitting workstation where *KitTrays* are placed to build *Kits*.
- A set of *LargeBoxWithKits*: A *LargeBoxWithKits* contains only finished *Kits*.
- A set of *LargeBoxWithEmptyKitTrays*: A *LargeBoxWithEmptyKitTrays* is a box that contains only empty *KitTrays*.
- A set of *Robot* {*robot_1*,*robot_2*,...}: A *Robot* in the kitting workstation is a robotic arm that can move objects in order to build *Kits*.
- A set of *EndEffector*: *EndEffectors* are used in a kitting workstation to manipulate *Parts*, *PartsTrays*, *KitTrays*, and *Kits*. An *EndEffector* is attached to a *Robot* in order to grasp objects.
- A set of *EndEffHolder*: An *EndEffHolder* is a storage unit that holds one type of *EndEffector*.
- A set of *EndEffChStation*: An *EndEffChStation* is made up of *EndEffHolders*.

2.2 Object Variable Symbols

Object variable symbols are typed variables which range over a class or the union of classes of constant variable symbols. Examples of object variable symbols are $r \in$ *Robots*, $kt \in$ *KitTrays*, etc.

2.3 State Variable Symbols

A state variable symbol is defined as follows:
$\mathrm{x} : \mathrm{A_1} \times \cdots \times \mathrm{A_i} \times \mathrm{S} \rightarrow \mathrm{B_1} \cup \cdots \cup \mathrm{B_j} \cup$ *bool* $\cup \{\} \cup$ *numeric* $(i, j \geq 1)$ is a function from the set of states (S) and at least one set of constant variable symbols $\mathrm{A_1} \times \cdots \times \mathrm{A_i}$ into a set $\mathrm{B_1} \cup \cdots \cup \mathrm{B_j} \cup$ *bool* $\cup \{\} \cup$ *numeric* where:

- $\mathrm{B_1} \cup \cdots \cup \mathrm{B_j}$ is a set of constant variable symbols
- *bool* is a boolean
- $\{\}$ is an empty set
- *numeric* is a numerical value

The use of state variable symbols reduces the possibility of inconsistent states and generates a smaller state space. The following state variable symbols are used in the kitting domain.

- endeffector-location
  EndEffector$\times$S $\rightarrow$*Robot* $\cup$ *EndEffHolder*: designates the location of an *EndEffector* in the workstation. An *EndEffector* is either attached to a *Robot* or placed in an *EndEffHolder*.
- robot-with-endeffector
  Robot$\times$S $\rightarrow$*EndEffector* $\cup$ $\{\}$: designates the *EndEffector* attached to a *Robot* if there is one attached, otherwise nothing.

- **on-worktable**
  $\mathit{WorkTable} \times \mathrm{S} \rightarrow \mathit{Kit} \cup \mathit{KitTray} \cup \{\}$: designates the object placed on the *WorkTable*, i.e., a *Kit*, a *KitTray*, or nothing.

- **kit-location**
  $\mathit{Kit} \times \mathrm{S} \rightarrow \mathit{LargeBoxWithKits} \cup \mathit{WorkTable} \cup \mathit{Robot}$: designates the different possible locations of a *Kit* in the workstation, i.e., in a *LargeBoxWithKits*, on the *WorkTable*, or being held by a *Robot*.

- **kittray-location**
  $\mathit{KitTray} \times \mathrm{S} \rightarrow \mathit{LargeBoxWithEmptyKitTrays} \cup \mathit{WorkTable} \cup \mathit{Robot}$: designates the different possible locations of a *KitTray* in the workstation, i.e., in a *LargeBoxWithEmptyKitTrays*, on a *WorkTable* or being held by a *Robot*.

- **part-location**
  $\mathit{Part} \times \mathrm{S} \rightarrow \mathit{PartsTray} \cup \mathit{Kit} \cup \mathit{Robot}$: designates the different possible locations of a *Part* in the workstation, i.e., in a *PartsTray*, in a *Kit*, or being held by a *Robot*.

- **robot-holds**
  $\mathit{Robot} \times \mathrm{S} \rightarrow \mathit{KitTray} \cup \mathit{Kit} \cup \mathit{Part} \cup \{\}$: designates the object being held by a *Robot*, i.e., a *KitTray*, a *Kit*, a *Part*, or nothing. It is assumed that the *Robot* is already equipped with the appropriate *EndEffector*.

- **lbwk-full**
  $\mathit{LargeBoxWithKits} \times \mathrm{S} \rightarrow \mathit{bool}$: designates if a *LargeBoxWithKits* is full or not.

- **lbwekt-empty**
  $\mathit{LargeBoxWithEmptyKitTrays} \times \mathrm{S} \rightarrow \mathit{bool}$: designates if a *LargeBoxWithEmptyKitTrays* is empty or not.

- **partstray-empty**
  $\mathit{PartsTray} \times \mathrm{S} \rightarrow \mathit{bool}$: designates if a *PartsTray* is empty or not.

- **endeffector-type**
  $\mathit{EndEffector} \times \mathrm{S} \rightarrow \mathit{KitTray} \cup \mathit{Kit} \cup \mathit{Part}$: designates the type of object an *EndEffector* can hold, i.e., a *KitTray*, a *Kit*, or a *Part*.

- **endeffectorholder-holds-endeff**
  $\mathit{EndEffHolder} \times \mathrm{S} \rightarrow \mathit{EndEffector} \cup \{\}$: designates wether an *EndEffHolder* is holding an *EndEffector* or nothing.

- **endeffectorholder-location**
  $\mathit{EndEffHolder} \times \mathrm{S} \rightarrow \mathit{EndEffChStation}$: designates the *EndEffChStation* where the *EndEffHolder* is located.

- **endeffectorchangingstation-has-endeffholder**
  $\mathit{EndEffChStation} \times \mathrm{S} \rightarrow \mathit{EndEffHolder}$: designates the *EndEffHolder* the *EndEffChStation* contains.

- **found-part**
  $\mathit{PartsTray} \times \mathrm{S} \rightarrow \mathit{Part} \cup \{\}$: designates wether a *Part* is found in a *PartsTray* or not.

- origin-part
  Part×S →*PartsTray*: designates the *PartsTray* where the *Part* is found.

- quantity-parts-in-partstray
  PartsTray×S →*numeric*: designates the number of parts that *PartsTray* contains.

- quantity-parts-in-kit
  Kit×*PartsTray*×S →*numeric*: designates the number of parts from *PartsTray* that *Kit* contains.

- capacity-parts-in-kit
  Kit×*PartsTray*×S →*numeric*: designates the number of parts from *PartsTray* that *Kit* **can** contain.

2.4 Predicates and Functions

In PDDL, predicates are used to encode Boolean state variables, while functions are used to model updates of numerical values [5]. This section describes the predicates and functions derived from the state variables described in section 2.3. We recall the following definition of a state variable ((section 2.3)) $\mathrm{x} : \mathrm{A}_1 \times \cdots \times \mathrm{A}_\mathrm{i} \times \mathrm{S} \rightarrow \mathrm{B}_1 \cup \cdots \cup \mathrm{B}_\mathrm{j}$ $(i, j \geq 1)$ that is used to convert state variables into predicates as follows:

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

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

2.4.1 Predicates

The state variables in our current kitting domain contains the following predicates.

- endeffector-location
  1. endeffector-location-robot(*EndEffector*,*Robot*)
     - ☐ TRUE iff *EndEffector* is attached to *Robot*
  2. endeffector-location-endeffholder(*EndEffector*,*EndEffHolder*)
     - ☐ TRUE iff *EndEffector* is in *EndEffHolder*

- robot-with-endeffector
  3. robot-with-endeffector(*Robot*,*EndEffector*)
     - ☐ TRUE iff *Robot* is equipped with *EndEffector*
  4. robot-with-no-endeffector(*Robot*)

  - ☐ TRUE iff *Robot* is not equipped with any *EndEffector*

- ■ on-worktable

5. on-worktable-kit(*WorkTable*,*Kit*)
   - ☐ TRUE iff *Kit* is on the *WorkTable*
6. on-worktable-kittray(*WorkTable*,*KitTray*)
   - ☐ TRUE iff *KitTray* is on the *WorkTable*
7. worktable-empty(*WorkTable*)
   - ☐ TRUE iff there is nothing on the *WorkTable*

- ■ kit-location

8. kit-location-lbwk(*Kit*,*LargeBoxWithKits*)
   - ☐ TRUE iff *Kit* is in the *LargeBoxWithKits*
9. kit-location-worktable(*Kit*, *WorkTable*)
   - ☐ TRUE iff *Kit* is on the *WorkTable*
10. kit-location-robot(*Kit*,*Robot*)
    - ☐ TRUE iff *Kit* is being held by the *Robot*

- ■ kittray-location

11. kittray-location-lbwekt(*KitTray*,*LargeBoxWithEmptyKitTrays*)
    - ☐ TRUE iff *KitTray* is in the *LargeBoxWithEmptyKitTrays*
12. kittray-location-robot(*KitTray*,*Robot*)
    - ☐ TRUE iff *KitTray* is being held by the *Robot*
13. kittray-location-worktable(*KitTray*, *WorkTable*)
    - ☐ TRUE iff *KitTray* is on the *WorkTable*

- ■ part-location

14. part-location-partstray(*Part*,*PartsTray*)
    - ☐ TRUE iff *Part* is in the *PartsTray*
15. part-location-kit(*Part*,*Kit*)
    - ☐ TRUE iff *Part* is in the *Kit*
16. part-location-robot(*Part*,*Robot*)
    - ☐ TRUE iff *Part* is being held by the *Robot*

- ■ robot-holds

17. robot-holds-kittray(*Robot*,*KitTray*)
    - ☐ TRUE iff *Robot* is holding a *KitTray*
18. robot-holds-kit(*Robot*,*Kit*)
    - ☐ TRUE iff *Robot* is holding a *Kit*
19. robot-holds-part(*Robot*,*Part*)

  - ☐ TRUE iff *Robot* is holding a *Part*

20. robot-empty(*Robot*)
    - ☐ TRUE iff *Robot* is not holding anything

- ■ lbwk-full

21. lbwk-not-full(*LargeBoxWithKits*)
    - ☐ TRUE iff *LargeBoxWithKits* is not full

- ■ lbwekt-empty

22. lbwekt-not-empty(*LargeBoxWithEmptyKitTrays*)
    - ☐ TRUE iff *LargeBoxWithEmptyKitTrays* is not empty

- ■ partstray-empty

23. partstray-not-empty(*PartsTray*)
    - ☐ TRUE iff *PartsTray* is not empty

- ■ endeffector-type

24. endeffeffector-type-kittray(*EndEffector*,*KitTray*)
    - ☐ TRUE iff *EndEffector* is capable of holding a *KitTray*
25. endeffector-type-kit(*EndEffector*,*Kit*)
    - ☐ TRUE iff *EndEffector* is capable of holding a *Kit*
26. endeffector-type-part(*EndEffector*,*Part*)
    - ☐ TRUE iff *EndEffector* is capable of holding a *Part*

- ■ endeffectorholder-holds-endeff

27. endeffectorholder-holds-endeff(*EndEffHolder*,*EndEffector*)
    - ☐ TRUE iff *EndEffHolder* is holding *EndEffector*
28. endeffectorholder-empty(*EndEffHolder*)
    - ☐ TRUE iff *EndEffHolder* is empty (not holding an *EndEffector*)

- ■ endeffectorholder-location

29. endeffectorholder-location(*EndEffHolder*,*EndEffChStation*)
    - ☐ TRUE iff *EndEffHolder* is in *EndEffChStation*

- ■ endeffectorchangingstation-has-endeffholder

30. endeffectorchangingstation-has-endeffholder(*EndEffChStation*,*EndEffHolder*)
    - ☐ TRUE iff *EndEffChStation* contains *EndEffHolder*

- ■ found-part

31. found-part(*Part*,*PartsTray*)
    - ☐ TRUE iff *Part* is found in *PartsTray*

- ■ origin-part

32. origin-part(*Part*,*PartsTray*)
    - ☐ TRUE iff *Part* is from *PartsTray*

2.4.2 Functions

In a planning model, numeric fluents represent function symbols that can take an infinite set of values. Introducing functions into planning not only makes it possible to deal with numerical values in a more general way than allowed for by a purely relational language but makes it possible to model operators in a more compact and sometimes also more natural way. The state variables in our current kitting domain contains the following functions.

- quantity-parts-in-partstray
  - quantity-parts-in-partstray(*PartsTray*)
    - Number of parts in *PartsTray*
- quantity-parts-in-kit
  - quantity-parts-in-kit(*Kit*,*PartsTray*)
    - Number of parts from *PartsTray* that is in *Kit*
- capacity-parts-in-kit
  - capacity-parts-in-kit(*Kit*,*PartsTray*)
    - Number of parts from *PartsTray* that *Kit* **can** contain

2.5 Planning Operators and Actions

The planning operators presented in this section are expressed in classical representation instead of state variable representation. In classical representation, states are represented as sets of logical atoms (predicates) that are true or false within some interpretation. Actions are represented by planning operators that change the truth values of these atoms.

2.5.1 Planning Operators

In classical planning, a planning operator [7] is a triple o=(*name*(o), *preconditions*(o), *effects*(o)) whose elements are as follows:

- *name*(o) is a syntactic expression of the form $n(x_1, \ldots, x_k)$, where n is a symbol called an operator symbol, $x_1, \ldots, x_k$ are all of the object variable symbols that appear anywhere in o, and n is unique (i.e., no two operators can have the same operator symbol).
- *preconditions*(o) and *effects*(o) are sets of literals (i.e., atoms and negations of atoms). Literals that are true in *preconditions*(o) but false in *effects*(o) are removed by using negations of the appropriate atoms.

Our kitting domain is composed of ten operators which are defined below.

1. *take-kittray*(*robot*,*kittray*,*lbwekt*,*endeff*,*worktable*): The *Robot robot* equipped with the *EndEffector endeff* picks up the *KitTray kittray* from the *LargeBoxWithEmptyKitTrays lbwekt*.

preconditions	*effects*
robot-empty(*robot*)	¬robot-empty(*robot*)
kittray-location-lbwekt(*kittray*,*lbwekt*)	¬kittray-location-lbwekt(*kittray*,*lbwekt*)
lbwekt-not-empty(*lbwekt*)	kittray-location-robot(*kittray*,*robot*)
robot-with-endeff(*robot*,*endeff*)	robot-holds-kittray(*robot*,*kittray*)
endeffector-location-robot(*endeff*,*robot*)	
worktable-empty(*worktable*)	
endeffector-type-kittray(*endeff*,*kittray*)	

- *preconditions*
    - robot-empty(*robot*): *robot* does not hold anything.
    - kittray-location-lbwekt(*kittray*,*lbwekt*): *kittray* is in *lbwekt*.
    - lbwekt-not-empty(*lbwekt*): *lbwekt* is not empty (contains at least one kit tray).
    - robot-with-endeffector(*robot*,*endeff*): *robot* is equipped with *endeff*.
    - endeffector-location-robot(*endeff*,*robot*): The end effector is on the robot's arm.
    - worktable-empty(*worktable*): After picking up an empty kit tray from a large box of empty kit trays, the robot would normally place the kit tray on the work table. To put a kit tray on the work table, it is necessary that there is nothing on top of the work table. If the robot is allowed to pick up the kit tray while there is another object on the work table, the planning system may not be able to find a solution when it comes to put the kit tray on the work table. Therefore, it is necessary to check that the top of *worktable* is clear even before the robot picks up a kit tray from the large box of empty kit trays.
    - endeffector-type-kittray(*endeff*,*kittray*): *endeff* in the robot's arm must be capable of handling *kittray*.
- *effects*
    - ¬robot-empty(*robot*): *robot*'s end effector is no longer empty since it contains *kittray*.
    - ¬kittray-location-lbwekt(*kittray*,*lbwekt*): *kittray* is no longer in *lbwekt* since it is in the robot's end effector.
    - kit-tray-location(*kittray*,*robot*): *kittray* is in *robot*'s end effector.
    - robot-holds-kittray(*robot*,*kittray*): *robot* is holding *kittray*.

2. *put-kittray*(*robot*,*kittray*,*worktable*): The *Robot robot* puts the *KitTray kittray* on the *WorkTable worktable*.

preconditions	*effects*
kittray-location-robot(*kittray*,*robot*)	¬kittray-location-robot(*kittray*,*robot*)
robot-holds-kittray(*robot*,*kittray*)	¬robot-holds-kittray(*robot*,*kittray*)
worktable-empty(*worktable*)	¬worktable-empty(*worktable*)
	kittray-location-wtable(*kittray*,*worktable*)
	robot-empty(*robot*)
	on-wtable-kittray(*worktable*,*kittray*)

- *preconditions*
    - kittray-location-robot(*kittray*,*robot*): *kittray* is in *robot*'s end effector.
    - robot-holds-kittray(*robot*,*kittray*): *robot* holds *kittray*.
    - worktable-empty(*worktable*): There is nothing on *worktable*.
- *effects*

- ☐ ¬kittray-location-robot(*kittray*,*robot*): *kittray* is no longer it *robot*'s end effector since it is placed on *worktable*.
- ☐ ¬robot-holds-kittray(*robot*,*kittray*): *robot* is not holding *kittray* anymore.
- ☐ ¬worktable-empty(*worktable*): *worktable* is not empty anymore since there is something on top of it.
- ☐ kittray-location-wtable(*kittray*,*worktable*): *kittray* is on *worktable*.
- ☐ robot-empty(*robot*): *robot* is not holding anything.
- ☐ on-wtable-kittray(*worktable*,*kittray*): *worktable* has *kittray* on top of it.

3. take-kit(*robot*,*kit*,*worktable*,*endeff*): The *Robot robot* equipped with the *EndEffector endeff* picks up the *Kit kit* from the *WorkTable worktable*.

preconditions	*effects*
kit-location-wtable(*kit*,*worktable*)	¬kit-location-wtable(*kit*,*worktable*)
robot-empty(*robot*)	¬robot-empty(*robot*)
on-wtable-kit(*worktable*,*kit*)	¬on-wtable-kit(*worktable*,*kit*)
robot-with-endeff(*robot*,*endeff*)	kit-location-robot(*kit*,*robot*)
endeff-type-kit(*endeff*,*kit*)	robot-holds-kit(*robot*,*kit*)
	worktable-empty(*worktable*)

- ■ *preconditions*
  - ☐ kit-location-wtable(*kit*,*worktable*): *kit* is located on *worktable*.
  - ☐ robot-empty(*robot*): *robot* is not holding any object.
  - ☐ on-wtable-kit(*worktable*,*kit*): *worktable* has *kit* on top of it.
  - ☐ robot-with-endeff(*robot*,*endeff*): *robot* is equipped with *endeff*.
  - ☐ endeff-type-kit(*endeff*,*kit*): The type of *endeff* is capable of handling *kit*.
- ■ *effects*
  - ☐ ¬kit-location-wtable(*kit*,*worktable*): *kit* is not on *worktable*.
  - ☐ ¬robot-empty(*robot*): *robot* is holding an object (*kit*).
  - ☐ ¬on-wtable-kit(*worktable*,*kit*): *worktable* does not have *kit* on top of it.
  - ☐ kit-location-robot(*kit*,*robot*): *kit* is being held by *robot*.
  - ☐ robot-holds-kit(*robot*,*kit*): *robot* is holding *kit*.
  - ☐ worktable-empty(*worktable*): *worktable* does not have any object on top of it.

4. put-kit(*robot*,*kit*,*lbwk*): The *Robot robot* puts down the *Kit kit* in the *LargeBoxWithKits lbwk*.

preconditions	*effects*
kit-location-robot(*kit*,*robot*)	¬kit-location-robot(*kit*,*robot*)
robot-holds-kit(*robot*,*kit*)	¬robot-holds-kit(*robot*,*kit*)
lbwk-not-full(*lbwk*)	kit-location-lbwk(*kit*,*lbwk*)
	robot-empty(*robot*)

- ■ *preconditions*
  - ☐ kit-location-robot(*kit*,*robot*): *kit* is held by *robot*.
  - ☐ robot-holds-kit(*robot*,*kit*): *robot* is holding *kit*.
  - ☐ lbwk-not-full(*lbwk*): *lbwk* should not be full so it can contain *kit*.
- ■ *effects*
  - ☐ ¬kit-location-robot(*kit*,*robot*): *kit* is not being held by *robot*.

- ☐ ¬robot-holds-kit(*robot*,*kit*): *robot* is not holding *kit*.
- ☐ kit-location-lbwk(*kit*,*lbwk*): *kit* has been placed in *lbwk*.
- ☐ robot-empty(*robot*): *robot* is not holding anything (not holding *kit* anymore).

5. *look-for-part*(*robot*,*part*,*partstray*,*kit*,*worktable*,*endeff*): A sensor looks for the *Part part* in the *PartTray partstray*.

preconditions	*effects*
part-not-searched	¬part-not-searched
robot-empty(*robot*)	found-part(*partstray*)
robot-with-endeff(*robot*,*endeff*)	
on-wtable-kit(*worktable*,*kit*)	
endeff-location-robot(*endeff*,*robot*)	
part-location-partstray(*part*,*partstray*)	
kit-location-wtable(*kit*,*worktable*)	
endeff-type-part(*endeff*,*part*)	
partstray-not-empty(*partstray*)	

- ■ *preconditions*
  - ☐ part-not-searched: This flag is set to true in the initial state in the problem file (see section 4.2) and means that a part has not been searched yet.
  - ☐ robot-empty(*robot*): *robot* should not be holding anything. We want the operator *look-for-part* to be directly followed by the operator *take-part* to simulate a sensor identifying a part before being picked up by a robot. It is necessary to check that *robot*'s end effector is empty to prepare for the execution of the operator *take-part*.
  - ☐ robot-with-endeff(*robot*,*endeff*): *robot* is equipped with *endeff*. Again, since we want the operator following *look-for-part* to be *take-part*, we want to make sure that *robot* is already equipped with *endeff*.
  - ☐ on-wtable-kit(*worktable*,*kit*): *worktable* has *kit* on top of it.
  - ☐ endeff-location-robot(*endeff*,*robot*): *endeff* is on *robot* so it is ready for the operator *take-part*.
  - ☐ part-location-partstray(*part*,*partstray*): *part* is in *partstray*.
  - ☐ kit-location-wtable(*kit*,*worktable*): *kit* is on *worktable*.
  - ☐ endeff-type-part(*endeff*,*part*): *endeff* can handle *part*.
  - ☐ partstray-not-empty(*partstray*): *partstray* contains at least one part.
- ■ *effects*
  - ☐ ¬part-not-searched: This flag is set to true so that the operator *look-for-part* can be called again to look for another part in the workstation.
  - ☐ found-part(*partstray*): A part from *partstray* has been found.

6. *take-part*(*robot*,*part*,*partstray*,*endeff*,*worktable*,*kit*): The *Robot robot* uses the *EndEffector endeff* to pick up the *Part part* from the *PartTray partstray*.

preconditions	*effects*
part-location-partstray(*part*,*partstray*)	¬part-location-partstray(*part*,*partstray*)
robot-empty(*robot*)	¬robot-empty(*robot*)
endeff-location-robot(*endeff*,*robot*)	part-location-robot(*part*,*robot*)
robot-with-endeff(*robot*,*endeff*)	robot-holds-part(*robot*,*part*)
on-wtable-kit(*worktable*,*kit*)	decrease quantity-parts-in-partstray(*partstray*)
kit-location-wtable(*kit*,*worktable*)	
endeff-type-part(*endeff*,*part*)	
partstray-not-empty(*partstray*)	
found-part(*part*,*partstray*)	

- *preconditions*
  - part-location-partstray(*part*,*partstray*): *part* to be picked up is in *partstray*.
  - robot-empty(*robot*): *robot* is not holding any object.
  - endeff-location-robot(*endeff*,*robot*): *endeff* is on *robot*.
  - robot-with-endeff(*robot*,*endeff*): *robot* is equipped with *endeff*.
  - on-wtable-kit(*worktable*,*kit*): *worktable* has *kit* on top of it.
  - kit-location-wtable(*kit*,*worktable*): *kit* is on *worktable*. Once a part is picked up by the robot, the next logical action would be to put the part in the kit. For this to happen, the kit needs to be already on the work table so it can hold the part.
  - endeff-type-part(*endeff*,*part*): *endeff* is the type for *part* handling.
  - partstray-not-empty(*partstray*): *partstray* is not empty and contains at least one part.
  - found-part(*part*,*partstray*): *part* has been found in *partstray*. found-part is set to true in the *effects* of the operator look-for-part.
- *effects*
  - ¬part-location-partstray(*part*,*partstray*): *part* is not in *partstray* anymore since it was picked up by *robot*.
  - ¬robot-empty(*robot*): *robot* is now holding *part* and is not empty anymore.
  - part-location-robot(*part*,*robot*): *part* is held by *robot*.
  - robot-holds-part(*robot*,*part*): *robot* is holding *part*.
  - decrease quantity-parts-in-partstray(*partstray*): After picking up a part from *partstray* the number of parts in *partstray* is decreased by one. This is expressed with the decrease function.

7. put-part(*robot*,*part*,*kit*,*worktable*,*partstray*): The *Robot robot* puts the *Part part* in the *Kit kit*.

preconditions	*effects*
part-location-robot(*part*,*robot*)	¬part-location-robot(*part*,*robot*)
robot-holds-part(*robot*,*part*)	¬robot-holds-part(*robot*,*part*)
on-wtable-kit(*worktable*,*kit*)	¬found-part(*part*,*partstray*)
kit-location-wtable(*kit*,*worktable*)	robot-empty(*robot*)
origin-part(*part*,*partstray*)	part-location-kit(*part*,*kit*)
($<$ (quantity-parts-in-kit(*kit*,*partstray*))	(increase (quantity-kit(*kit*,*partstray*)))
(capacity-kit(*kit*,*partstray*)))	part-not-searched

- *preconditions*
  - part-location-robot(*part*,*robot*): *part* is held by *robot*.
  - robot-holds-part(*robot*,*part*): *robot* is holding *part*.

- ☐ on-wtable-kit(*worktable*,*kit*): *worktable* has *kit* on top of it.
- ☐ kit-location-wtable(*kit*,*worktable*): *kit* is on *worktable*.
- ☐ origin-part(*part*,*partstray*): *part* is from *partstray*. This is used to tell the type of *part*.
- ☐ (< (quantity-kit(*kit*,*partstray*)) (capacity-kit(*kit*,*partstray*))): The quantity of parts of type *partstray* in *kit* should be lesser than the capacity *kit* can hold for this type of part.

■ *effects*

- ☐ ¬part-location-robot(*part*,*robot*): *part* is not held by *robot*.
- ☐ ¬robot-holds-part(*robot*,*part*): *robot* is not holding *part*.
- ☐ robot-empty(*robot*): *robot* is not holding any object.
- ☐ part-location(*part*,*kit*): *part* is located in *kit*.
- ☐ (increase (quantity-kit(*kit*,*partstray*))): Once *part* is placed in *kit*, the quantity of *parts* in *kit* is increased by one.
- ☐ part-not-searched: This flag is set to true so another part search (through the operator *look-for-part*) is made after *part* is placed in *kit*.

8. *attach-endeff*(*robot*,*endeff*,*endeffholder*,*endeffchstation*): The *Robot robot* attaches the *EndEffector endeff* which is situated in the *EndEffHolder endeffholder*.

preconditions
endeff-location-endeffholder(*endeff*,*endeffholder*)
robot-with-no-endeff(*robot*)
endeffholder-holds-endeff(*endeffholder*,*endeff*)
endeffholder-location(*endeffholder*,*endeffchstation*)
endeffchstation-contains-endeffholder(*endeffchstation*,*endeffholder*)

effects
¬endeff-location-endeffholder(*endeff*,*endeffholder*)
¬endeffholder-holds-endeff(*endeffholder*,*endeff*)
¬robot-with-no-endeff(*robot*)
robot-empty(*robot*)
endeff-location-robot(*endeff*,*robot*)
robot-with-endeff(*robot*,*endeff*)
endeffholder-empty(*endeffholder*)

■ *preconditions*

- ☐ endeff-location-endeffholder(*endeff*,*endeffholder*): *endeff* is located in *endeffholder*.
- ☐ robot-with-no-endeff(*robot*): *robot* is not equipped with any *endeff*.
- ☐ endeffholder-holds-endeff(*endeffholder*,*endeff*): *endeffholder* is holding *endeff*.
- ☐ endeffholder-location(*endeffholder*,*endeffchstation*): *endeffholder* is in *endeffchstation*.
- ☐ endeffchstation-contains-endeffholder(*endeffchstation*,*endeffholder*): *endeffchstation* contains *endeffholder*.

■ *effects*

- ☐ ¬endeff-location-endeffholder(*endeff*,*endeffholder*): *endeff* is not in *endeffholder* anymore since it has been attached to *robot*.
- ☐ ¬endeffholder-holds-endeff(*endeffholder*,*endeff*): *endeffholder* is not holding *endeff* anymore.
- ☐ ¬robot-with-no-endeff(*robot*): *robot* is now equipped with *endeff*.
- ☐ robot-empty(*robot*): *robot* is not holding any object.
- ☐ endeff-location-robot(*endeff*,*robot*): *endeff* is on *robot*.
- ☐ robot-with-endeff(*robot*,*endeff*): *robot* is equipped with *endeff*.

- ☐ endeffholder-empty(*endeffholder*): *endeffholder* is not holding any *endeff*.

9. remove-endeff(*robot*,*endeff*,*endeffholder*,*endeffchstation*): The *Robot robot* removes the *EndEffector endeff* and puts it in the *EndEffHolder endeffholder*.

preconditions
endeff-location-robot(*endeff*,*robot*)
robot-with-endeff(*robot*,*endeff*)
robot-empty(*robot*)
endeffholder-location(*endeffholder*,*endeffchstation*)
endeffchstation-contains-endeffholder(*endeffchstation*,*endeffholder*)
endeffholder-empty(*endeffholder*)

effects
¬endeff-location-robot(*endeff*,*robot*)
¬robot-with-endeff(*robot*,*endeff*)
¬endeffholder-empty(*endeffholder*)
endeff-location-endeffholder(*endeff*,*endeffholder*)
endeffholder-holds-endeff(*endeffholder*,*endeff*)
robot-with-no-endeff(*robot*)

- ■ *preconditions*
  - ☐ endeff-location-robot(*endeff*,*robot*): *endeff* is on *robot*.
  - ☐ robot-with-endeff(*robot*,*endeff*): *robot* is holding *endeff*.
  - ☐ robot-empty(*robot*): *robot* is not holding anything.
  - ☐ endeffholder-location(*endeffholder*,*endeffchstation*): *endeffholder* is in *endeffchstation*.
  - ☐ endeffchstation-contains-endeffholder(*endeffchstation*,*endeffholder*)
  - ☐ endeffholder-empty(*endeffholder*): *endeffholder* does not contain *endeff*.
- ■ *effects*
  - ☐ ¬endeff-location-robot(*endeff*,*robot*): *endeff* is not on *robot* anymore.
  - ☐ ¬robot-with-endeff(*robot*,*endeff*): *robot* does not have *endeff* anymore.
  - ☐ ¬endeffholder-empty(*endeffholder*): *endeffholder* does not contain any *endeff*.
  - ☐ endeff-location-endeffholder(*endeff*,*endeffholder*): *endeff* is situated in *endeffholder*.
  - ☐ endeffholder-holds-endeff(*endeffholder*,*endeff*): *endeffholder* holds *endeff*.
  - ☐ robot-with-no-endeff(*robot*): *robot* is not equipped with *endeff*.

10. create-kit(*kit*,*kittray*,*worktable*): The *KitTray kittray* is converted into the *Kit kit* once the *KitTray kittray* is on the *WorkTable worktable*.

preconditions	*effects*
on-wtable-kittray(*worktable*,*kittray*)	¬on-wtable-kittray(*worktable*,*kittray*)
	on-wtable-kit(*worktable*,*kit*)
	kit-location-wtable(*kit*,*worktable*)

- ■ *preconditions*
  - ☐ on-wtable-kittray(*worktable*,*kittray*): *worktable* has *kittray* on top of it.
- ■ *effects*
  - ☐ ¬on-wtable-kittray(*worktable*,*kittray*): The object *kittray* is destroyed and is thus not on *worktable* anymore.
  - ☐ on-wtable-kit(*worktable*,*kit*): *worktable* now has *kit* on top of it.
  - ☐ kit-location-wtable(*kit*,*worktable*): *kit* is on *worktable*.

2.5.2 Actions

An action a can be obtained by substituting the object variable symbols that appear anywhere in the operator with constant variable symbols. For instance, the operator take-part(*robot*,*part*,*partstray*,*endeff*) in the kitting domain can be translated into the action take-part(*robot_1*,*part_1*,*partstray_1*,*endeff_2*) where *robot_1*, *part_1*, *partstray_1*, and *endeff_2* are constant variable symbols in the classes *Robot*, *Part*, *PartsTray*, and *EndEffector*, respectively.

3 Ontology

Once the state variable representation defined, an expert builds a knowledge representation for the kitting domain. As depicted in Figure 1(page 1), the kitting workstation model has been fully defined in each of two languages: XML schema language [?], [?], [?], and Web Ontology Language (OWL) [?], [?], [?].

In order to maintain compatibility with the IEEE working group, the ontology has been fully defined in OWL. However, due to several difficulties defined below, the ontology was also fully defined in the XML schema language. Although the two models are conceptually identical, there are some systematic differences between the models (in addition to differences inherent in using two different languages).

- The complexType names (i.e., class names) in XML schema have the suffix "Type" added which is not used in OWL. This is so that the same names without the suffix can be used in XML schema language as element names without confusion.
- All of the XML schema complexTypes have a "Name" element that is not present in OWL. It is not needed in OWL because names are assigned as a matter of course when instances of classes are created.
- The XML schema model has a list of "Object" elements. This collects all of the movable objects. The OWL model does not have a corresponding list. In an OWL data file, the movable objects may appear anywhere.
- Attribute names in OWL have a prefix, as described below. The prefixes are not used in XML schema.

3.1 OWL Specifics

The kitting workstation model was defined first in OWL because the IEEE RAS Ontologies for Robotics and Automation Working Group has decided to use OWL, and the authors are participating in the activities of that working group. OWL allows the use of several different syntaxes. The functional-style syntax (which is the most compact one) has been used to write the OWL version of the kitting workstation model.

In addition to having the model defined in OWL, OWL data files describing specific initial states and goal states were defined in OWL, also using the functional-style syntax. Software tools were built in C++ and Java to work with the OWL model and data files conforming to the model.

The initial intent has been to use OWL files for presenting the initial and goal conditions for planning problems, and the authors have implemented a planning system that uses OWL files.

The primary tool used by the OWL community for building and checking OWL models and data files is named Protégé [?]. Protégé was used for checking the kitting model and data files as they were built. Protégé continues to be used for checking the model and data files whenever they are changed.

Defining a model in OWL is quite different from defining the same model in other information modeling languages with which the authors are intimately familiar: C++, EXPRESS [?], and

XML schema. Three of the major differences involve (1) the assignment of attributes in classes, (2) OWL's "open world" assumption, and (3) the distinction between model files and data files.

3.1.1 Class Attributes

In other languages, assigning a typed attribute to a class requires a single line of code. For example, the X attribute may be put into a cartesian point class in XML schema language with
<xs:element name="X" type="xs:decimal"/>
or in C++ with
double X;
or in EXPRESS with
X : REAL;
In these other languages, the name of the attribute is local to the class. Hence, an attribute with a given name can appear in more than one class, and there will be no confusion.

In OWL, there is no simple method of declaring a class attribute. Instead, a property must be declared along with properties of the property. The following lines are used in the OWL model to say that all points and only points have an X attribute which is a decimal number.

```
Declaration(DataProperty(hasPoint_X))
DataPropertyDomain(:hasPoint_X :Point)
DataPropertyRange(:hasPoint_X xsd:decimal)
EquivalentClasses(:Point ObjectIntersectionOf(
    DataSomeValuesFrom(:hasPoint_X xsd:decimal)
    DataAllValuesFrom(:hasPoint_X xsd:decimal)))
```

The hasPoint prefix used in the property name is not an OWL requirement. It is one of several naming conventions for OWL being used by the authors. The prefix is both for the benefit of a human reader (to make it obvious that this is a property of a Point) and to differentiate this X attribute from an X attribute of some other class (call it Foo) which would have the prefix hasFoo .

As described above, with OWL it is necessary to make many statements in order to build a class in a typical object-oriented style. OWL does not assume a typical object-oriented style. It assumes the world might be more complex than that. Hence, many OWL statements are required to produce effects made in a few statements in other object-oriented languages. Having to write a lot of statements is tedious but not a roadblock. A more serious problem is that if a statement necessary to produce an object-oriented effect is omitted, that is not an OWL error. Protégé does not have an object-oriented mode in which it will warn the user if a required statement is missing. There are no OWL tools that will help with finding missing statements. This is a debugging problem.

OWL was built so that it would support automated reasoning about the relationships among properties, classes, and individuals. Protégé allows the use of several alternate automatic reasoners. In a typical object-oriented style, there is no use for reasoning of that sort. Everything useful to know about the relationships among properties, classes, and individuals is already known. Hence having an automated reasoning capability of the sort for which OWL was built is not useful for the kitting model.

3.1.2 Open World Assumption

OWL makes an "open world" assumption. In an open world, anything might be true that is not explicitly declared false and is not inconsistent with what has been declared true. This makes it easy for errors to go unrecognized as such by Protégé (or any other OWL tool). For example, suppose the line DataPropertyDomain(:hasPoint_X :Point) given above is mistyped as DataProperty-Domain(:hasPoint_x :Point). When Protégé loads the file and the reasoner is started, no errors are detected. Protégé assumes that the DataPropertyDomain for hasPoint_X is unknown (that is not an error in OWL and Protégé) and that there is a new property named hasPoint_x about which the only thing known is its DataPropertyDomain (also not an error in OWL and Protégé, even though there is no explicit DataProperty declaration for the new property). The error can be detected by a human by studying the list provided by selecting the DataProperties tab in Protégé. Similar errors, such as mistyping the name of an individual, are similarly accepted without error in OWL and Protégé, with similar effects. The difficulties caused by the open world assumption would not occur if Protégé had a closed world mode, but it has none.

3.1.3 Model Files vs. Data Files

While other languages have different file formats for models and data conforming to the models, OWL does not distinguish between model files and data files. Protégé does not provide any method of specifying that a file is a model file or a data file. The conceptual difference is simple. Model files describe classes and data types (and, possibly, constraints). Data files give information about individuals (instances of one or more classes – often called objects). The authors have made it a practice to distinguish OWL model files from OWL data files. An OWL data file can inadvertently change an OWL model, a bug that is very hard to find. That cannot happen with EXPRESS or XML schema.

3.1.4 Bugs in Files

Since humans are error-prone, and the kitting OWL files were built by humans, the OWL files had errors of the sort mentioned above. Some of these errors were discovered when the OWL files were processed by the tools developed for processing them and strange results were observed. Other errors were found when a method of generating OWL data files automatically from XML data files was developed, as described next.

3.2 XML Specifics

To better explore the pros and cons of various representations, the authors are using XML schema and XML data files in parallel with the corresponding OWL files.

3.2.1 XML Tools

Two automated tools developed by the authors are being used: an xml schema parser (xmlSchema-Parser) and a code generator (GenXMiller).

The xmlSchemaParser reads an XML schema file, stores it in terms of instances of C++ classes, and reprints the schema. When the xmlSchemaParser runs, it performs many checks on the validity of the schema that is input to it. The xmlSchemaParser handles almost all portions of the XML schema syntax. A few of the rarely-used elements of syntax are not implemented.

The GenXMiller reads an XML schema and writes code for reading and writing XML data files corresponding to that schema. The code that is generated includes C++ classes (.hh and .cc files), a parser (YACC and Lex files), and a stand-alone parser file in C++ that uses the other files. The executable utility produced by compiling a stand-alone parser reads and echoes any XML data file corresponding to the schema. The GenXMiller is still under development and currently handles only a subset of the XML schema language. The GenXMiller is not a new type of system. Several other code generators that use an XML schema as input have been developed [**?**, **?**]. Even more XML schema parsers are available. However, having the knowledge about XML schema and XML data files gained by developing that software and having an intimate knowledge of the source code for it has proved very valuable in converting XML representations to OWL representations.

The xmlSchemaParser and the GenXMiller use the same underlying parser, which is built in YACC and Lex [**?**].

In addition to using the xmlSchemaParser and the GenXMiller, a commercial XML tool named XMLSpy [**?**] has been used to check all XML schemas and XML data files.

3.2.2 Handling Kitting Data Files

There is only one conceptual kitting model, but there are several kitting data files corresponding to it. If the kitting model is used to represent various starting and goal configurations, there will be many more data files. Hence, the problem of generating bug-free data files was tackled first.

An XML schema, kitting.xsd, was written by hand modeling the same information as the OWL kitting workstation model, kittingClasses.owl. The GenXMiller was then used to generate C++ classes and a parser for XML kitting data files corresponding to kitting.xsd. The C++ classes that were generated included code for printing XML kitting data files. That code was rewritten by hand so that it prints OWL data files rather than XML data files. The utility produced by compiling the code is called the owlPrinter. To produce an OWL kitting data file, one writes an XML kitting data file and runs it through the owlPrinter.

To determine that the owlPrinter works properly, it seems sufficient to demonstrate that OWL data files generated automatically by the owlPrinter from XML data files conforming to kitting.xsd contain exactly the same OWL statements as are contained in manually prepared OWL data files intended to contain the same information and conforming to kittingClasses.owl. This demonstration was achieved as follows.

(i) Three XML data files were written manually containing the same information as three OWL data files. Each of the OWL files was at least 1,100 lines (20 pages) long. Among the three there were statements of almost all of the types possible under the kittingClasses.owl model. It was decided, therefore, that successful performance for these three files would be an adequate test.

(ii) The three XML data files were run through the owlPrinter to produce three OWL files.

(iii) Since the owlPrinter has a different approach to ordering OWL statements than was taken in preparing OWL files manually, and a slightly different method of formatting statements, two small utilities were written to enable file comparison. The first utility, compactOwl, reads an OWL file and writes an OWL file containing the same statements but with blank lines and comments removed, and with each statement on a single line. For each pair of matching OWL files (manually written and automatically generated), compactOwl was used to generate a corresponding pair of compacted OWL files. The second utility, compareOwl, reads each of a pair of OWL files, alphabetizes the statements from each of them on two saved lists, and then goes through the two lists checking that the nth line of one list is identical to the nth line of the other list. CompareOwl was used to compare each of the three sets of pairs of compacted files.

(iv) While the tests just described were being made, changes were made to correct errors in the manually written XML and OWL data files being tested and in the code for the owlPrinter. The tests revealed errors in all three types of files.

After the testing just described was complete, using the owlPrinter another OWL data file was prepared from a manually written XML data file for which there was no manually written OWL counterpart. The automatically generated OWL data file was checked in Protégé and no errors were reported.

OWL data files may now be prepared with much less likelihood of human error for the following reasons.

- Property names and names of individuals will not be misspelled.
- Statements will not be accidentally omitted.
- Validity checks made in the kittingParser and XMLSpy will do a better job of detecting errors in XML data files. For example, required attributes that are missing will be detected.

3.2.3 Handling the Kitting Model

As described above, the equivalent model files kitting.xsd and kittingClasses.owl were both prepared manually. If changes to the kitting model are made, it will be necessary to change both of those files and the code for the owlPrinter. It would be good to have kitting.xsd as the primary source file for the model and to generate kittingClasses.owl automatically from it. The authors believe this is possible and have started working on it. The work is not yet complete, but no roadblocks are anticipated. The approach being using is to modify the printer code in the xmlSchemaParser so that it prints an OWL class file rather than an XML schema file.

It would also be desirable to be able to modify the owlPrinter automatically if the kitting model is changed. Doing that is a substantially more difficult task than the other two automatic conversions, and the authors are not planning to attempt it. The approach would be to modify the GenXMiller so that the code it generates automatically would read XML data files and automatically generate OWL data files.

3.3 Tools

This section describes the tools that are used to generate OWL files from XML files.

3.3.1 owlPrinter

owlPrinter reads an XML kitting data file corresponding to an XML schema for kitting (kitting.xsd) and writes an OWL instance file corresponding to the OWL class file kittingClasses.owl. The kitting.xsd file contains the same conceptual model as the kittingClasses.owl file, but in a different language.

The owlPrinter is useful because there is no OWL tool that will help generate an OWL instance file and check the file adequately against an OWL class file. That is because OWL uses an open world model in which anything not explicitly or implicitly illegal is allowed. Hence many things that are errors to the writer of the instance file are not OWL errors. For example, if the name of an instance is misspelled, OWL will assume that there is a new instance that has not been explicitly declared as such, which is OK in OWL. If a reference to an instance name is misspelled in an XML data file corresponding to the kitting.xsd schema, that will be caught automatically by the owlPrinter (and other readily available XML tools). Several other types of error will not be caught by OWL tools but will not be made or will be detected if the OWL printer is used.

Another OWL problem that disappears in XML is that in OWL, there is no distinction between an instance file and a class file. An instance file can modify classes, intentionally, or accidentally. In XML there is no way a data file can modify a model.

To use the owlPrinter, use a text editor such as emacs or an XML tool such as XMLSpy to write an XML data file corresponding to the kitting.xsd schema and then run it through the owlPrinter with a command of the form:

```
bin/owlPrinter [XML file in] [OWL file out]
```

For example, the command

```
bin/owlPrinter data/kittingInstances.xml junk
```

will print the file junk, which will be identical to the kittingInstances.owl file in the owl directory (except for a couple comments).

3.3.2 kittingParser

The kittingParser may be used to check an XML data file against the kitting.xsd schema. The schema is hard-coded into the kittingParser. If there is any error in the XML data file, the kittingParser prints a message and quits. If there is no error, the input file is echoed by printing an output file whose name is the same as that of the input file with "echo" appended. To run the kittingParser, give a command of the form:

```
bin/kittingParser [XML kitting data file in]
```

For example, the command

```
bin/kittingParser data/kittingInstances.xml
```

will read kittingInstances.xml and write kittingInstances.xmlecho. The two files will be identical except for comments. If the format of an input file differs from the format used for printing the output file, the two files will differ, but only in format.

The owlPrinter makes the same checks as the kittingParser, so there is no need to use the kittingParser.

All of the source code for the kittingParser was generated automatically by the GenXMiller generator.

3.3.3 compactOwl and compareOwl

There is no need to read this section unless you are interested in how the *owlPrinter* was debugged.

The *compactOwl* and *compareOwl* utilities are used for checking that two different OWL instance files have the same statements. They have been used as follows to debug the *owlPrinter* (and the kitting.xsd file and the kittingInstances.xml file and the kittingInstances.owl file).

1. Write kitting.xsd to model the same information as kittingClasses.owl.
2. Write kittingInstances.xml to correspond to kitting.xsd and contain the same information as kittingInstances.owl.
3. Build the *owlPrinter*.
4. Run kittingInstances.xml through the *owlPrinter* to produce kittingInstances.owl.
5. Run kittingInstances.owl through *compactOwl* to produce one version of kittingInstancesCompact.owl.
6. Run kittingInstances.owl through *compactOwl* to produce a second version of kittingInstancesCompact.owl.
7. Run the two versions of kittingInstancesCompact.owl through *compareOwl*. If *compareOwl* reports that the two files have the same statements, that means that steps 1, 2, and 3 have been done correctly, so debugging is finished. If *compareOwl* reports a pair of statements that differ, figure out why, go back to step 1, 2, or 3 (or edit kittingInstances.owl), fix the problem, and repeat the subsequent steps.

These utilities assume that the format of the input files is the same as either the format used by the *owlPrinter* or the format followed by the kittingInstances.owl file. If the format used by an input file is different from both of those, the utilities may fail.

The *compactOwl* utility compacts an OWL instance file by:

1. removing all occurrences of one or two blank lines. The blank lines must not contain spaces or tabs.
2. removing comments. The comments must have "//" as the first two characters on the line.
3. combining each OWL statement written on two or more lines so it is all on one line. The first non-space character on the second line must be a colon (:) or a double quote (").
4. rewriting numbers with decimal points so there are exactly six decimal places. Such numbers must have at least one digit on each side of the decimal point in the input file.
5. putting the **DifferentIndividuals** inside **DifferentIndividuals** statements into alphabetical order.

To run *compactOwl* use a command of the form:

```
bin/compactOwl < [owl file in] > [owl file out]
```

where `[owl file in]` and `[owl file out]` are replaced by file names.

The *compareOwl* utility compares two files that are expected to have the same lines, but in a different order, such as an automatically generated OWL file and a hand-written OWL file. It reads the two files and saves the lines of each one in two sets in alphabetical order (set::insert puts strings in alphabetical order by default). Then it compares the lines of the two sets in order. If it finds two lines that do not match, it prints the line from the first file followed by the line from the second file. If all lines match, that is reported.

To run *compareOwl*, use a command of the form:

```
bin/compareOwl [first owl file in] [second owl file in]
```

where `[first owl file in]` and `[second owl file in]` are replaced by the names of compacted OWL files.

4 Planning Language

The Planning Domain Definition Language (PDDL) [6] is an attempt by the domain independent planning community to formulate a standard language for planning. A community of planning researchers has been producing planning systems that comply with this formalism since the first International Planning Competition held in 1998. This competition series continues today, with the seventh competition being held in 2011. PDDL is constantly adding extensions to the base language in order to represent more expressive problem domains. Our work is based on PDDL Version 3.
By placing our knowledge in a PDDL representation, we enable the use of an entire family of open source planning systems. Each PDDL file-set consists of two files that specify the domain and the problem.

4.1 The PDDL Domain File

The PDDL domain file for kitting is consists of five sections that include `requirements`, `types`, `predicates`, `functions`, and `actions`. An excerpt of the PDDL domain file is depicted in Figure 2.

```
 1. (define (domain kitting-domain)
 2.     (:requirements :strips :typing :fluents)
 3.     (:types EndEffector EndEffectorHolder Kit KitTray LargeBoxWithEmptyKitTrays
 4.         PartsTray EndEffectorChangingStation Robot WorkTable LargeBoxWithKits Part)
 5.
 6.     (:predicates
 7.            (endeffector-location-robot ?endeffector - EndEffector ?robot - Robot)
 8.            (on-worktable-kit ?worktable - WorkTable ?kit - Kit))
 9.
10.     (:functions
11.            (quantity-partstray ?partstray - PartsTray)
12.            (quantity-kit ?kit - Kit ?partstray - PartsTray)
13.            (capacity-kit ?kit - Kit ?partstray - PartsTray))
14.
15.     (:action take-kittray
16.         :parameters(
17.             ?robot - Robot
18.             ?kittray - KitTray
19.             ?largeboxwithemptykittrays - LargeBoxWithEmptyKitTrays
20.             ?endeffector - EndEffector
21.             ?worktable - WorkTable)
22.         :precondition(and
23.             (robot-empty ?robot)
24.             (lbwekt-not-empty ?largeboxwithemptykittrays)
25.             (robot-with-endeffector ?robot ?endeffector)
26.             (kittray-location-lbwekt ?kittray ?largeboxwithemptykittrays)
27.             (endeffector-location-robot ?endeffector ?robot)
28.             (worktable-empty ?worktable)
29.             (endeffector-type-kittray ?endeffector ?kittray))
30.         :effect(and
31.             (robot-holds-kittray ?robot ?kittray)
32.             (kittray-location-robot ?kittray ?robot)
33.             (not (robot-empty ?robot))
34.             (not (kittray-location-lbwekt ?kittray ?largeboxwithemptykittrays))))
35. )
36.
```

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

- 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.

- line 2: The `:requirements` field specifies which section the domain relies on. The planning system can examine this statement to determine if it is capable of solving problems in this domain. A keyword (symbol starting with a colon) used in a `:requirements` field is called a requirement flag; the domain is said to declare a requirement for that flag. The requirement flags present in the kitting domain are:
  - `:strips`: The most basic subset of PDDL, consisting of STRIPS only.
  - `:typing`: PDDL has a special syntax for declaring parameter and object types. `:typing` allows types names in declaration of variables.
  - `:fluents`: A domain's set of requirements allow a planner to quickly tell if it is likely to be able to handle the domain. For example, this version of the kitting world requires fluents numeric, so a straight STRIPS-representation planner would not be able to handle it. A fluent is a term (`:functions`) with time-varying value (i.e., a value that can change as a result of performing an action).
- lines 3–4: Type names have to be declared before they are used (before `:predicates` and `:functions`). This is done with the declaration `(:types` $name_1$ `...` $name_n$`)`.
- lines 6–8: 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.
  It is common to make a distinction between static and dynamic predicates. A *static* predicate is not changed by any action. Thus in a problem, the true and false instances of a *static* predicate will always be precisely those listed in the initial state specification of the problem definition. Note that there is no syntactic difference between *static* and *dynamic* predicates in PDDL, they look exactly the same in the `:predicates` declaration part of the domain.
  A predicate is build using the structure `(predicate_name ?X - type_of_X)`. A list of parameters of the same type in a predicate can be abbreviated to `(predicate_name ?X ?Y ?Z - type_of_XYZ)`. Note that the hyphen between parameter and type name is surrounded by whitespace.
- lines 10–13: A fluent is similar to a state variable/predicate except that its value is a number instead of true or false. The initial value of a function is set in the initial state of the problem file and changes when an action is executed. The declaration of functions is similar to predicates.
- lines 15–34: 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. All parts of an action definition except the name are, according to the PDDL specification, optional (although, of course, an action without effects is pretty useless). However, for an action that has no preconditions some planners may require an "empty" precondition, on the form :precondition () or :precondition (and), and some planners may also require an empty :parameter list for actions without parameters).
  - lines 16–21: The `:parameters` section declare all the parameters used by predicates and functions in `preconditions` and `effects`.

- ☐ lines 22–29: The `:preconditions` section is a conjunction of predicates and functions that need to be true in the world in order for the action to be invoked.
- ☐ lines 30–34: The `:effects` equation dictates the changes in the world that will occur due to the execution of the action.

4.2 PDDL Problem File

The second file of the PDDL file-set is a problem file. The problem file specifies information about the specific instance of the given problem. This file contains the initial conditions and definition of the world (in the `init` section) and the final state that the world must be brought to (in the `goal` section). Using an example of kit to build, this section only describes the initial and goal states explicitly. The operators detailed in Section 2.5 are used by a planner to generate the other states as needed.
In the PDDL problem file depicted below, the *Robot* has to build a kit that contains two *Parts* of type A, one *Part* of type B and one *Part* of type C. The kitting process is completed once the *Kit* is placed in the *LargeBoxWithKits*.

```
(define (problem kitting-problem)
    (:domain kitting-domain)
    (:objects
        robot_1 - Robot
        changing_station_1 - EndEffectorChangingStation
        kit_tray_1 - KitTray
        kit_a2b1c1 - Kit
        empty_kit_tray_supply - LargeBoxWithEmptyKitTrays
        finished_kit_receiver - LargeBoxWithKits
        work_table_1 - WorkTable
        part_a_tray part_b_tray part_c_tray - PartsTray
        part_a_1 part_a_2  - Part
        part_b_1 part_b_2 part_b_3 - Part
        part_c_1 part_c_2 - Part
        part_gripper tray_gripper - EndEffector
        part_gripper_holder tray_gripper_holder - EndEffectorHolder
    )
)
(:init
    (robot-with-no-endeffector robot_1)
    (part-not-searched)
    (lbwekt-not-empty empty_kit_tray_supply)
    (lbwk-not-full finished_kit_receiver)
    (partstray-not-empty part_a_tray)
    (partstray-not-empty part_b_tray)
    (partstray-not-empty part_c_tray)
    (endeffector-location-endeffectorholder part_gripper part_gripper_holder)
    (endeffector-location-endeffectorholder tray_gripper tray_gripper_holder)
    (endeffectorholder-holds-endeffector part_gripper_holder part_gripper)
    (endeffectorholder-holds-endeffector tray_gripper_holder tray_gripper)
    (endeffectorholder-location tray_gripper_holder changing_station_1)
    (endeffectorholder-location part_gripper_holder changing_station_1)
    (endeffectorchangingstation-contains-endeffectorholder changing_station_1 tray_gripper_holder)
    (endeffectorchangingstation-contains-endeffectorholder changing_station_1 part_gripper_holder)
    (worktable-empty work_table_1)
    (kittray-location-lbwekt kit_tray_1 empty_kit_tray_supply)

38.
39.     (part-location-partstray part_a_1 part_a_tray)
40.     (part-location-partstray part_a_2 part_a_tray)
41.     (part-location-partstray part_b_1 part_b_tray)
42.     (part-location-partstray part_b_2 part_b_tray)
43.     (part-location-partstray part_b_3 part_b_tray)
44.     (part-location-partstray part_c_1 part_c_tray)
45.     (part-location-partstray part_c_2 part_c_tray)
46.
47.     (endeffector-type-part part_gripper part_a_1)
48.     (endeffector-type-part part_gripper part_a_2)
49.     (endeffector-type-part part_gripper part_b_1)
50.     (endeffector-type-part part_gripper part_b_2)
51.     (endeffector-type-part part_gripper part_b_3)
52.     (endeffector-type-part part_gripper part_c_1)
53.     (endeffector-type-part part_gripper part_c_2)
54.
55.     (endeffector-type-kittray tray_gripper kit_tray_1)
56.     (endeffector-type-kit tray_gripper kit_a2b1c1)
57.
58.     (= (capacity-kit kit_a2b1c1 part_a_tray) 2)
59.     (= (capacity-kit kit_a2b1c1 part_b_tray) 1)
60.     (= (capacity-kit kit_a2b1c1 part_c_tray) 1)
61.
62.     (= (quantity-kit kit_a2b1c1 part_a_tray) 0)
63.     (= (quantity-kit kit_a2b1c1 part_b_tray) 0)
64.     (= (quantity-kit kit_a2b1c1 part_c_tray) 0)

65.     (= (quantity-partstray part_a_tray) 2)
66.     (= (quantity-partstray part_b_tray) 3)
67.     (= (quantity-partstray part_c_tray) 2)
68.
69.     (origin-part part_a_1 part_a_tray)
70.     (origin-part part_a_2 part_a_tray)
71.     (origin-part part_b_1 part_b_tray)
72.     (origin-part part_b_2 part_b_tray)
73.     (origin-part part_b_3 part_b_tray)
74.     (origin-part part_c_1 part_c_tray)
75.     (origin-part part_c_2 part_c_tray)
76. )
77.
78. (:goal
79.     (and
80.         (= (quantity-kit kit_a2b1c1 part_a_tray)(capacity-kit kit_a2b1c1 part_a_tray))
81.         (= (quantity-kit kit_a2b1c1 part_b_tray)(capacity-kit kit_a2b1c1 part_b_tray))
82.         (= (quantity-kit kit_a2b1c1 part_c_tray)(capacity-kit kit_a2b1c1 part_c_tray))
83.         (kit-location-lbwk kit_a2b2c1 finished_kit_receiver)
84.     )
85. )
```

- line 1: Signal a planner that the file contains all the element part of a problem. `kitting-problem` is the name given to this problem.
- line 2: `:domain` refers to the domain that the current problem is associated to. In this case, the problem refers to the domain `kitting-domain`. Note that `kitting-domain` is the name given to the kitting domain as presented in section 4.1.
- line 3–17: `:objects` declare objects present in the problem instance. The syntax for `:objects` is $object_1$ `- Type ...` $object_n$ `- Type`.

4.2.1 Initial State

The initial state S_0 (Figure 3) defines the environment in its initial condition. The initial state of the kitting problem in PDDL format is described below.

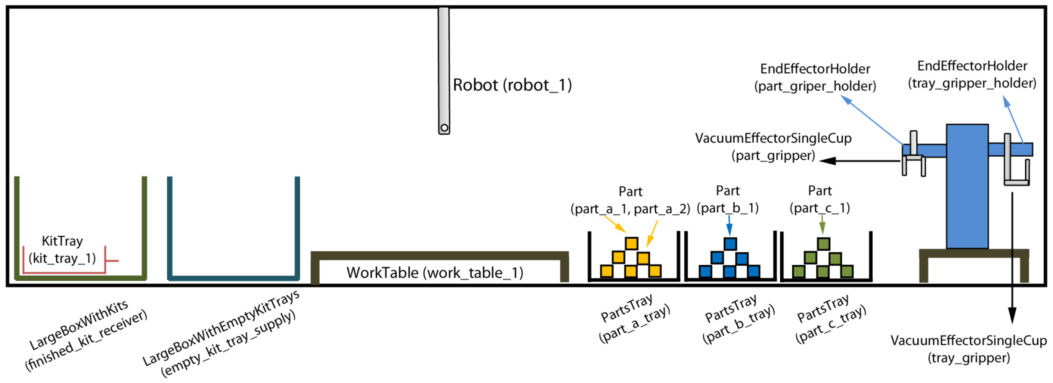

Robot (robot_1)
EndEffectorHolder (part_griper_holder)
EndEffectorHolder (tray_gripper_holder)
VacuumEffectorSingleCup (part_gripper)
Part (part_a_1, part_a_2)
Part (part_b_1)
Part (part_c_1)
KitTray (kit_tray_1)
WorkTable (work_table_1)
LargeBoxWithKits (finished_kit_receiver)
LargeBoxWithEmptyKitTrays (empty_kit_tray_supply)
PartsTray (part_a_tray)
PartsTray (part_b_tray)
PartsTray (part_c_tray)
VacuumEffectorSingleCup (tray_gripper)

Figure 3: Initial state S_0.

- line 19: `:init` signals a planner that the predicates and functions in this section are true in the initial state.
- line 20–76: Predicates true in the initial state of the environment. Since PDDL uses a close world assumption, predicates that are not present in the initial state are automatically set to false. This section also set the initial values for functions. Some relevant sections are presented:
  - line 21: The predicate part-not-searched is set to true so that the operator *look-for-part* can be activated during a plan search.
  - line 58–60: Functions describing how many parts of a specific type that *kit_a2b1c1* can contain. In this example, *kit_a2b1c1* can have two *Parts* of type A (*part_a_tray*), one *Part* of type B (*part_b_tray*), and one *Part* of type C (*part_c_tray*).
  - line 62–64: Functions that represent the number of parts of a specific type that are already in *kit_a2b1c1*. In the initial state, *kit_a2b1c1* is empty (no *Parts* of type A, B, or C).
  - line 65–67: Functions that describe the number of parts available in their respective parts tray. This also can be read as: *In the workstation, there are two Parts of type A available, three Parts of type B available, and three Parts of type C available.*
  - line 69–75: Predicates that describe the type of each specific part in the workstation. Defining that `part_a_1` is from `part_a_tray` is similar to `part_a_tray` is of type A since a *PartsTray* consists of parts of the same type.

4.2.2 Goal State

Figure 4 depicts the goal state S_G for the kitting workstation. The expression of the goal state in PDDL is described below.

- line 78: `:goal` is a keyword used to signal a planner about the goal state to reach. All the predicates and functions in the goal state must be true.

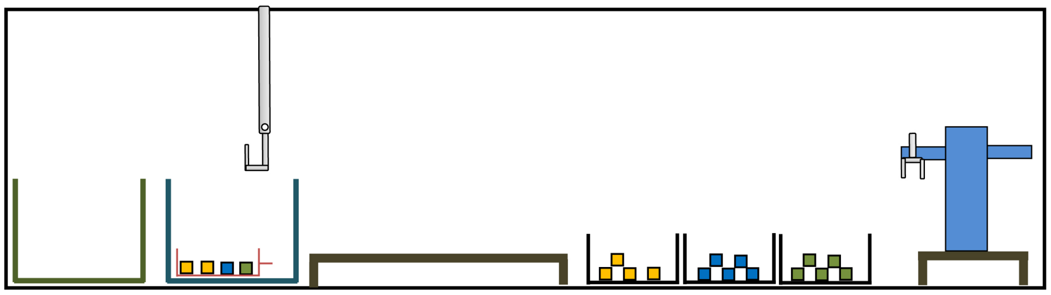


Figure 4: Goal state S_G.

- line 80–82: The quantity of parts of a specific type in *kit_a2b1c1* should match the capacity of parts of a specific type for *kit_a2b1c1*. The quantity of parts in *kit_a2b1c1* is increased in the operator *put-part*. The initial quantity of parts in *kit_a2b1c1* (lines 62–64) and its capacity (lines 58–60) are set in the initial state. Note that we are not specifying which instance of *Part* should go in *kit_a2b1c1* but rather the number of *Parts* of a specific type that *kit_a2b1c1* must have.
- line 83: *kit_a2b1c1* should be placed in the large box with kits *finished_kit_receiver*.

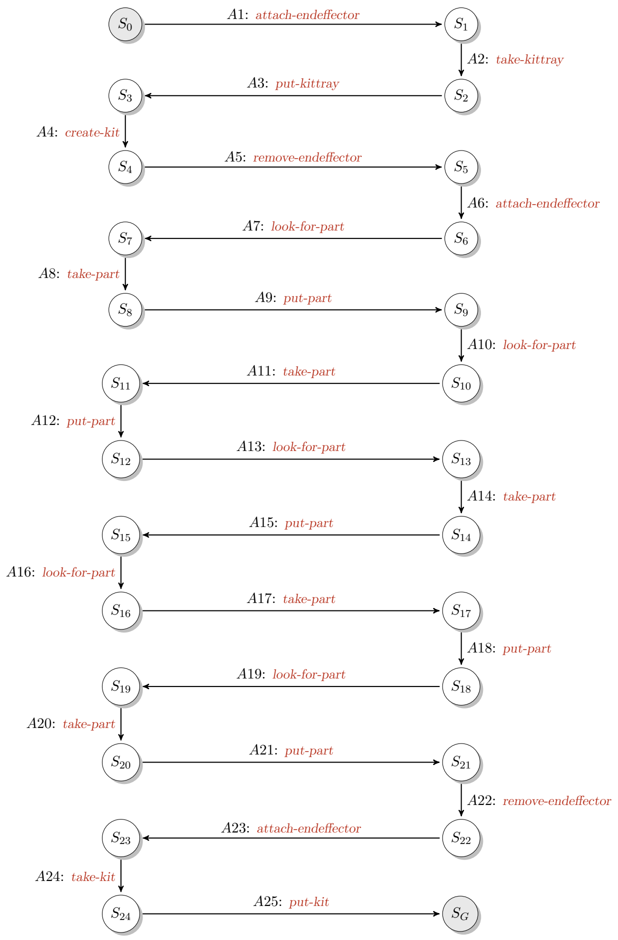
4.3 Plan

This section shows an example of a plan generated by a planner for the PDDL domain and problem files discussed previously in this document.

Figure 5 displays the different states and actions used by the planner to generate a plan starting from the initial state S_0 to the goal state S_G. The actions $A_1 \ldots A_{17}$ are described below.

- *A1*:(attach-endeffector *robot_1 tray_gripper tray_gripper_holder changing_station_1*)
- *A2*:(take-kittray *robot_1 kit_tray_1 empty_kit_tray_supply tray_gripper work_table_1*)
- *A3*:(put-kittray *robot_1 kit_tray_1 work_table_1*)
- *A4*:(create-kit *kit_a2b2c1 kit_tray_1 work_table_1*)
- *A5*:(remove-endeffector *robot_1 tray_gripper tray_gripper_holder changing_station_1*)
- *A6*:(attach-endeffector *robot_1 part_gripper part_gripper_holder changing_station_1*)
- *A7*:(look-for-part *robot_1 part_c_1 part_c_tray kit_a2b2c1 work_table_1 part_gripper*)
- *A8*:(take-part *robot_1 part_c_1 part_c_tray part_gripper work_table_1 kit_a2b2c1*)
- *A9*:(put-part *robot_1 part_c_1 kit_a2b2c1 work_table_1 part_c_tray*)
- *A10*:(look-for-part *robot_1 part_b_2 part_b_tray kit_a2b2c1 work_table_1 part_gripper*)
- *A11*:(take-part *robot_1 part_b_2 part_b_tray part_gripper work_table_1 kit_a2b2c1*)
- *A12*:(put-part *robot_1 part_b_2 kit_a2b2c1 work_table_1 part_b_tray*)
- *A13*:(look-for-part *robot_1 part_b_1 part_b_tray kit_a2b2c1 work_table_1 part_gripper*)

- *A14*:(take-part *robot_1 part_b_1 part_b_tray part_gripper work_table_1 kit_a2b2c1*)
- *A15*:(put-part *robot_1 part_b_1 kit_a2b2c1 work_table_1 part_b_tray*)
- *A16*:(look-for-part *robot_1 part_a_2 part_a_tray kit_a2b2c1 work_table_1 part_gripper*)
- *A17*:(take-part *robot_1 part_a_2 part_a_tray part_gripper work_table_1 kit_a2b2c1*)
- *A18*:(put-part *robot_1 part_a_2 kit_a2b2c1 work_table_1 part_a_tray*)
- *A19*:(look-for-part *robot_1 part_a_1 part_a_tray kit_a2b2c1 work_table_1 part_gripper*)
- *A20*:(take-part *robot_1 part_a_1 part_a_tray part_gripper work_table_1 kit_a2b2c1*)
- *A21*:(put-part *robot_1 part_a_1 kit_a2b2c1 work_table_1 part_a_tray*)
- *A22*:(remove-endeffector *robot_1 part_gripper part_gripper_holder changing_station_1*)
- *A23*:(attach-endeffector *robot_1 tray_gripper tray_gripper_holder changing_station_1*)
- *A24*:(take-kit *robot_1 kit_a2b2c1 work_table_1 tray_gripper*)
- *A25*:(put-kit *robot_1 kit_a2b2c1 finished_kit_receiver*)



$$\begin{tikzcd}
S_0 \arrow[rrr, "A1"] & & & S_1 \arrow[d, "A2"] \\
S_3 \arrow[d, "A4"'] & & & S_2 \arrow[lll, "A3"'] \\
S_4 \arrow[rrr, "A5"] & & & S_5 \arrow[d, "A6"] \\
S_7 \arrow[d, "A8"'] & & & S_6 \arrow[lll, "A7"'] \\
S_8 \arrow[rrr, "A9"] & & & S_9 \arrow[d, "A10"] \\
S_{11} \arrow[d, "A12"'] & & & S_{10} \arrow[lll, "A11"'] \\
S_{12} \arrow[rrr, "A13"] & & & S_{13} \arrow[d, "A14"] \\
S_{15} \arrow[d, "A16"'] & & & S_{14} \arrow[lll, "A15"'] \\
S_{16} \arrow[rrr, "A17"] & & & S_{17} \arrow[d, "A18"] \\
S_{19} \arrow[d, "A20"'] & & & S_{18} \arrow[lll, "A19"'] \\
S_{20} \arrow[rrr, "A21"] & & & S_{21} \arrow[d, "A22"] \\
S_{23} \arrow[d, "A24"'] & & & S_{22} \arrow[lll, "A23"'] \\
S_{24} \arrow[rrr, "A25"] & & & S_G
\end{tikzcd}$$

Figure 5: Example of plan generated with the kitting domain and problem files.

5 Robot Language

6 Planner

This section describes the steps to install and run a planner on the PDDL domain and problem files in order to generate a plan. The planner uses a forward-chaining partial-order planning [4].

6.1 Requirements

The planner requires:

- `cmake`
- The CBC mixed integer programming solver (`https://projects.coin-or.org/Cbc/`)
- `perl`, `bison` and `flex` to build the parser

These are packaged with most Linux distributions - on Ubuntu/Debian, the following should suffice:

```
sudo apt-get install cmake coinor-libcbc-dev coinor-libclp-dev \
coinor-libcoinutils-dev bison flex
```

6.2 Download and Install CBC

The CBC source code can be retrieved in two ways:

1. Local copy: `ipmas/planner/coin-Cbc.tar.gz`
2. SVN: `svn co https://projects.coin-or.org/svn/Cbc/stable/2.8 coin-Cbc`

If CBC is retrieved using option 1, unzip `coin-Cbc.tar.gz` to generate the `coin-Cbc` directory.

Perform the following steps.

- `cd coin-Cbc`
- `./configure -C`: Runs a configure script that generates the make file.
- `make`: Builds the Cbc library and executable program.
- `make test`: Builds and runs the Cbc unit test program.
- `make install`: Installs libraries, executables and header files in directories `coin-Cbc/lib`, `coin-Cbc/bin` and `coin-Cbc/include`.

6.3 Compile the Planner

The planner can be found at `ipmas/planner/popf2-11jun2011.tar.bz2`. Unzip `popf2-11jun2011.tar.bz2` to get the `tempo-sat-popf2` directory, then:

- cd `tempo-sat-popf2`
- `./build`

New files and directories are created in the `tempo-sat-popf2/compile/` directory. However, the executable is not generated at this point and errors should be displayed. To fix this, open `tempo-sat-popf2/compile/CMakeCache.txt` in a text editor and edit the following lines. Note that `<path>` is the absolute path that leads to the `coin-Cbc` directory.

- **line 24**: `CBC_INCLUDES:PATH=<path>/include`
- **line 33**: `CGL_INCLUDES:PATH=<path>/include`
- **line 36**: `CGL_LIBRARIES:FILEPATH=/usr/lib/libCgl.so.0`
- **line 39**: `CLP_INCLUDES:PATH=<path>/include`
- **line 186**: `COINUTILS_INCLUDES:PATH=<path>/include`
- **line 230**: `OSICLP_LIBRARIES:FILEPATH=/usr/lib/libOsiClp.so.0`
- **line 233**: `OSI_INCLUDES:PATH=<path>/include`
- **line 236**: `OSI_LIBRARIES:FILEPATH=/usr/lib/libOsi.so.0`

Recompile the planner:
`./build`

6.4 Run the Planner

To run the planner, the path to the PDDL domain and problem files should be identified. The format of the PDDL files must be `.pddl`. The following command run the planner on the PDDL files.

- `./plan <domain> <problem> <solution>`

Where `<domain>` and `<problem>` are the paths that point to the PDDL domain and problem files, respectively.
`<solution>` is the output file that will contain the plan.

7 Tools

7.1 The Generator

The Generator tool is a graphical user interface developed in Java, allowing the user to store data from OWL files into a MySQL database. This tool also permits the user to query the database using the C++ function calls. The tool Generator is composed of the following functionalities:

1. Convert OWL documents into SQL syntaxes (OWL to SQL).
2. Translate SQL syntaxes to OWL language in order to modify an OWL document (SQL to OWL).
3. Convert the OWL language into C++ classes (OWL to C++).

To date, only steps 1. and 3. have been implemented and will be covered in this document.

7.1.1 Prequisites

The description of the Generator tool is given for a Ubuntu Linux system. To run and use the Generator tool, different applications must be installed on the system.

Java Runtime Environment The Generator tool comes as a jar file. As such, the Java Runtime Environment should be installed on your system. This application can be found at `www.oracle.com`.

MySQL Server and Client The MySQL server and client should be installed and running on your system.

- *sudo apt-get update* (Update the package management tools)
- *sudo apt-get dist-upgrade* (Install the latest software)
- *sudo apt-get install mysql-server mysql-client* (Install the MySQL server and client packages). You will be asked to enter a password.

When done, you have a MySQL database ready to run. The following command will allow you to run MySQL.

- *mysql -u root -p*
- Enter the same password you used when you installed MySQL.

Finally, we need the plugin `libmysqlcppconn-dev` which allows C++ to connect to MySQL databases. It can be installed as follows:

- *sudo apt-get install libmysqlcppconn-dev*

7.1.2 How to Run the Generator Tool

The Generator tool can be launched using either one of these two following methods:

1. *java -jar Generator.jar*
2. Right-click on Generator.jar and select the option "Open With OpenJDK Java 6 Runtime". Note that this message will be different for future releases of the Java Runtime Environment.

7.1.3 Functionalities

As mentioned in the Introduction, we are covering only steps 1. and 3. in the rest of this document, i.e., *OWL to SQL* and *OWL to C++*, respectively.

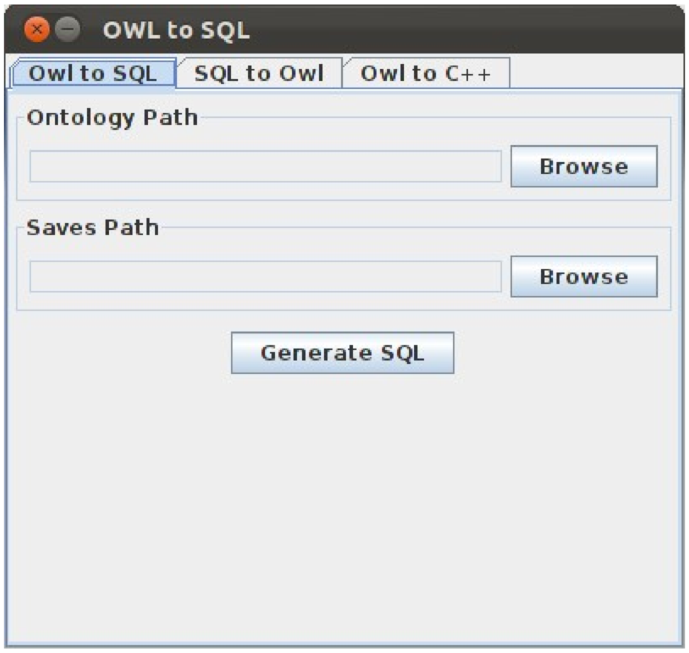


Figure 6: Owl to SQL tab.

OWL to SQL To convert OWL classes and instances to SQL, the `Owl to SQL` tab should be selected (see Figure 6). The different fields are:

Generate SQL Files

- Ontology Path: This field requires the file `kittingInstances.owl`. Before doing so, you need to modify one line in this file. Open it with a text editor and find the line `Import(<file:kittingClasses.owl>)`. Modify this line by giving the absolute path to the file `kittingClasses.owl`. You should should have something that looks like `Import(<file:/home/username/NIST/ipmas/Generator/kittingClasses.owl>)`. When this is done, save the file, and browse to `kittingInstances.owl` using the "Browse" button.

- Browse to the directory where you want to save the SQL files.

Once the two previous steps are done, click on "Generate SQL". You should receive a message confirming the generation of the SQL files: `kittingInstances.owlCreateTable.sql` and `kittingInstances.owlInsertInto.sql`. The former is used to create tables, the latter is used to populate these tables;

SQL Tables and Insertions The next step is to create a database and to populate it.

- Connect to mysql using *mysql -u root -p*, then enter your password. You should be in the mysql shell if this succeeded (*mysql>*).
- Delete a previous database (if you already used this tool and you want to replace the existing database with this new one) : `mysql>` *DROP DATABASE OWL;* (*OWL* is the name of the old database).
- Create a database:
  - `mysql>` *CREATE DATABASE OWL;*. Here, *OWL* is the name of the database (you can use a name of your choice).
  - Before performing the following commands, we need to tell MySQL which database we are planning to work with (*OWL* in our case). This is done using:

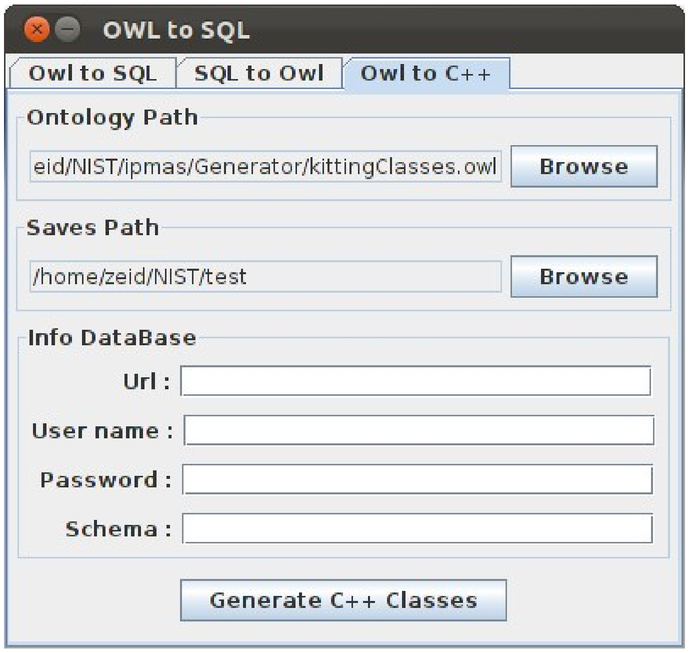
    `mysql>` *USE OWL*
- Populate the database with tables using `kittingInstances.owlCreateTable.sql`.
  - `mysql>` *source <path>/kittingInstances.owlCreateTable.sql;*
- Populate the tables with data using `kittingInstances.owlInsertInto.sql`:
  - `mysql>` *source <path>/kittingInstances.owlInsertInto.sql;*

<path> designs the absolute path to the appropriate file.

OWL to C++ The "Owl to C++" tab (see Figure 7) is used to generate C++ classes and scripts allowing the connection between C++ and MySQL. The different fields are explained below:

- **Ontology Path**: This is the path to the ontology (`kittingClasses.owl` in our example).
- **Saves Path**: Directory where the C++ files and scripts will be generated.
- **Url**: This is the url of the database. It's usually the IP address of the machine hosting the database (127.0.0.1 if it is local).
- **User name**: User name used to connect to the MySQL database.
- **Password**: Password associated to the user name to connect to the MySQL database.
- **Schema**: This is the name of the database (*OWL* in our example).

When all the fields are completed, click the "Generate C++ Classes" button to generate C++ and script files.



Ontology Path: eid/NIST/ipmas/Generator/kittingClasses.owl

Saves Path: /home/zeid/NIST/test

Info DataBase

Url :

User name :

Password :

Schema :

Figure 7: Owl to C++ tab.

7.2 XML to OWL

References

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

[2] S. Balakirsky, T. Kramer, and A. Pietromartire. Metrics and Test Methods for Industrial Kit Building. NISTIR to appear, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2012.

[3] S. Balakirsky, 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)*, pages 1365–1370, October 2012.

[4] A. J. Coles, A. Coles, M. Fox, and D. Long. Forward-Chaining Partial-Order Planning. In *20th International Conference on Automated Planning and Scheduling, ICAPS 2010*, pages 42–49, Toronto, Ontario, Canada, May 12–16 2010. AAAI 2010.

[5] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research (JAIR)*, 20(1):61–124, 2003.

[6] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL–The Planning Domain Definition Language. Technical Report CVC TR98-003/DCS TR-1165, Yale, 1998.

[7] D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

List of Figures