ICT Call 4
RoboEarth Project
2010-248942

Deliverable D5.2:

The RoboEarth Language -
Language Specification

August 2nd, 2010

Project acronym:	RoboEarth
Project full title:	Robots sharing a knowledge base for world modeling and learning of actions
Work package:	5
Document number:	D5.2
Document title:	The RoboEarth Language - Language Specification
Version:	1
Delivery date:	August 02, 2010
Nature:	Report
Dissemination level:	Public
Authors:	Moritz Tenorth (TUM) Michael Beetz (TUM)

The research leading to these results has received funding from the European Union Seventh Framework Programme FP7/2007-2013 under grant agreement n°248942 RoboEarth.

Summary

This document describes the current state of implementation of the RoboEarth representation language. This language is designed for two main purposes. First, it should allow to represent all information a robot needs to perform a reasonably complex task. This includes information about

- *Plans*, which consist of the actions a task is composed of, ordering constraints among them, monitoring and failure handling, as well as action parameters like objects, locations, grasp types;
- *Objects*, especially types, dimensions, states, and other properties, but also locations of specific objects a robot has detected, and object models that can be used for recognition; and the
- *Environment*, including maps for self-localization as well as poses of objects like pieces of furniture.

The second main task of the RoboEarth language is to allow a robot to decide on its own if a certain piece of information is useful to it. That means, a robot must be able to check if an action description contains a plan for the action it would like to do, if it meets all requirements to perform this action, and if it has the sensors needed to use an object recognition model. Using the semantic descriptions in the RoboEarth language, a robot can perform the checks using logical inference.

Chapter 1

Language Overview

This section introduces some of the main concepts of the RoboEarth language that are required for the remainder of this text. The language is implemented in OWL, the Web Ontology Language, which is an XML-based format for writing ontologies in Description Logics (DL).

1.1 Description Logics

Description logics are a family of logical languages for knowledge representation. There are several dialects with different expressiveness, most of which are a decidable subset of first-order logic. An extensive overview can be found in [1], a shorter introduction in [2]. Here, we will just briefly summarize the main concepts. Table 1.1 gives an overview of the DL syntax.

Description Logics distinguish between terminological knowledge, the so-called TBOX, and assertional knowledge, the ABOX. The TBOX contains definitions of concepts, for example the concepts *Action*, *SpatialThing*, *PickingUpAnObject* or *TableKnife*. These concepts are arranged in a hierarchy, a so-called taxonomy, using subclass definitions that state for instance that a *TableKnife* is a specialization of *SilverwarePiece*.

The ABOX contains individuals that belong to these concepts, e.g. *knife1* as an instantiation of the concept *TableKnife*. When modeling knowledge in robotics, the ABOX usually describes perceived things: detected object instances, observed actions, or perceived events. The TBOX, in contrast, describes classes of objects or actions.

Note that the differences between ABOX and TBOX are not domain specificness or environment dependency. There can be individuals in the ABOX that are very general as well as very specific classes in the TBOX, like the class of action "Grasping an egg from the refrigerator with the right hand using a pinch grasp".

Roles can describe the properties of an individual, describe the relation between two individuals, and can also be used in concept definitions to restrict the extent of a class to individuals having certain properties. For example, the concept *OpeningABottle* can be described as a subclass of *OpeningSomething* with the restriction that the *objectActedOn* has to be some instance of a *Bottle*.

$$OpeningABottle \sqsubseteq OpeningSomething \sqcap \exists objectActedOn.Bottle$$

This kind of knowledge representation, consisting of a set of concepts and relations between these concepts, is called an "ontology". The knowledge is formally represented and allows to draw conclusions using logical inference. In terms of expressiveness, DL subsumes UML class diagrams or entity-relationship models, two commonly used knowledge representation formalisms.

1.2 Web Ontology Language

The RoboEarth language is implemented in the Web Ontology Language (OWL), an XML-based language with direct correspondence to Description Logic[1]. Therefore, tools for reasoning on description logics can be used for inference on OWL ontologies. The nomenclature in OWL and DL differ slightly: "concepts" are usually called "classes" in OWL, "roles" are called "properties", and "individuals" are called "objects" or "instances". The following table compares the language constructs in DL and their correspondences in OWL (taken from [2]).

OWL provides the language constructs to represent knowledge about a domain, which is then encoded in terms of OWL ontologies. This can be of course done at very different levels of abstraction: There are upper ontologies that describe very abstract knowledge (e.g. general relations between actions, agents and objects), and there are domain-specific extensions, for example for household tasks or service robotics. The RoboEarth language is realized as such a domain-specific extension of an existing ontology: It is derived from the KnowRob ontology, a knowledge base describing the domain of household robots, and extends it with elements that are specific to the exchange of knowledge between robots. KnowRob itself is derived from the OpenCyc ontology, a very broad upper ontology, and adds knowledge specific to service robotics.

[1]The complete OWL reference can be found at http://www.w3.org/TR/owl-ref/

OWL Syntax	**DL Syntax**	**Example**
Thing	$\top$	
Nothing	$\bot$	
intersectionOf	$C_1 \sqcap \ldots \sqcap C_n$	$Human \sqcap Male$
unionOf	$C_1 \sqcup \ldots \sqcup C_n$	$Doctor \sqcup Lawyer$
complementOf	$\neg C$	$\neg Male$
oneOf	$\{x_1 \ldots x_n\}$	$\{john, mary\}$
allValuesFrom	$\forall r.C$	$\forall hasChild.Doctor$
someValuesFrom	$\exists r.C$	$\exists hasChild.Lawyer$
hasValue	$\exists r.x$	$\exists citizenOf.USA$
minCardinality	$(\leqslant\ n\,r)$	$(\leqslant\ 2\,hasChild)$
maxCardinality	$(\geqslant\ n\,r)$	$(\leqslant\ 1\,hasChild)$
inverseOf	r^-	$hasChild^-$
subClassOf	$C_1 \sqsubseteq C_2$	$Human \sqsubseteq Animal \sqcap Biped$
equivalentClass	$C_1 \equiv C_2$	$Man \equiv Human \sqcap Male$
subPropertyOf	$P_1 \sqsubseteq P_2$	$hasDaughter \sqsubseteq hasChild$
equivalentProperty	$P_1 \equiv P_2$	$cost \equiv price$
disjointWith	$C_1 \sqsubseteq \neg C_2$	$Male \sqsubseteq \neg Female$
sameAs	$\{x_1\} \equiv \{x_2\}$	$\{Pres\_Bush\} \equiv \{GW\_Bush\}$
differentFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$	$\{john\} \sqsubseteq \neg\{peter\}$
TransitiveProperty	$P\,transitive\,role$	$hasAncestor$
FunctionalProperty	$\top \sqsubseteq (\leqslant\ 1\,P)$	$\top \sqsubseteq (\leqslant\ 1\,hasMother)$
InverseFunctionalProperty	$\top \sqsubseteq (\leqslant\ 1\,P^-)$	$\top \sqsubseteq (\leqslant\ 1\,isMotherOf^-)$
SymmetricProperty	$P \equiv P^-$	$isSiblingOf \equiv isSiblingOf^-$

Table 1.1: Language elements in OWL and DL syntax. C_i are concepts, x_i are individuals, r,P_i are roles, and n is a positive integer. Examples taken from [2].

1.3 KnowRob knowledge processing system

The RoboEarth language builds upon the KnowRob[2] knowledge base [7] and uses some of KnowRob's reasoning capabilities. KnowRob is realized in a very modularized way. A base system provides the common ontology and reasoning methods and is largely independent of the domain of interest.

KnowRob is implemented in SWI Prolog using its Semantic Web library for loading loading, storing and reasoning on the OWL ontologies. When OWL files are loaded into the system, they are internally stored as triples rdf(Subject, Predicate, Object) and can be accessed with special predicates,

[2]Available for download: http://tum-ros-pkg.svn.sourceforge.net/viewvc/tum-ros-pkg/stacks/knowrob

e.g. rdf_has(S, P, O). These predicates operate on the internal representation that is created from the OWL files and handle properties like transitivity of properties etc. On the other hand, Prolog is used as a programming language to implement specialized reasoning modules and to interface the knowledge base with external data.

Extension modules plug into the system and provide e.g. specialized reasoning capabilities or interfaces to external data, e.g. to read object detections from the vision system. These modules mainly operate on the level of instances.

1.4 Links to external data

The RoboEarth language provides means to describe the semantics of the exchanged data, though on a rather abstract level. For many applications, however, there are already established and optimized file formats: Collada, for instance, is a widely used format for describing kinematics. Other modules, like object recognition systems, will have their own file formats that allow to efficiently store the required information. We try to keep these data formats to ensure compatibility and to profit from existing tools. The RoboEarth language thus allows to store links to these external data files. In this case, the actual information (an object model, an occupancy grid map, etc) is stored in an external file, and a description in the RoboEarth language adds meta-information like prerequisites that are required to use the model.

Chapter 2

RoboEarth central ontology

RoboEarth is intended to be a large, distributed system that can be used by various different robots, performing many different actions in very diverse environments. In such a setting, it is impossible to predict all kinds of actions or objects that need to be described. Instead, we need a flexible way to extend the language to new application domains.

At the same time, it is important to have a common basic language that describes the overall concepts (like the relations between actions, objects, grasps, and trajectories) that is the same for all communicating partners. Otherwise, a robot would not be able to understand a description it downloads from RoboEarth.

We therefore chose the following setup: A common central ontology describes the main concepts that are required. This ontology provides the language elements described in this report and supports the reasoning capabilities described in Section 4. This ontology can be extended by every user as described in the following section.

2.1 Extending the ontology

Extending the ontology means to define new classes, properties, and relations between classes in order to describe things that cannot be modeled with the existing base ontology. These classes and properties should be derived from the existing ones, so that a robot is able to link the new concepts to those it already knows.

For example, assume a robot would like to provide a recipe for a special kind of raisin cake, but there is only the general concept of *BakedThing* in the ontology. It should thus create the concept of *Cake* and *RaisinCake* by creating a sub-tree below *BakedThing*. Further, it can describe the properties like “a *RaisinCake* is a sweet *BakedThing* that contains raisins”. With this definition, another robot can use all its knowledge about *BakedThing*, and in addition to that classify observed raisin cakes as *RaisinCake* based on their properties.

These classes can be specified in the same OWL file that also describes the recipe. That is, the file with the recipe also provides a definition of the language used to describe the recipe. If a robot downloads the recipe, it thus receives both the extended language description and the recipe, which is describing actions using these new language elements.

Chapter 3

Representing action recipes in the RoboEarth language

In this chapter, we will explain how the RoboEarth language can be used to represent information on actions, objects, and the robot's environment in a way that robots can generate, exchange and use it. As a simple example, let's assume a robot has learned the task to navigate through an environment and to recognize and localize a cup.

To exchange the "recipe" for this action with another robot, it needs to encode different kinds of information: Task descriptions like the actions the task is composed of, how they need to be arranged, and which parameters are to be chosen, are to be combined with information about the environment, like a map for self-localization that may also describe where a suitable cup has already been found, and finally a description of the cup so that a robot that executes this action recipe knows it if has found the correct one. In the following sections, we will describe how such information can be encoded.

3.1 Meta data

In addition to the descriptions of actions, objects, and the environment, the language further needs to encode meta-data that is to help a robot assess the information and select the right one. For example, if there are different recipes for a task, the robot should have criteria to choose a good one. In general, it should only download information that it can make use of, i.e. for which it has the right prerequisites.

3.1.1 Information on data creation

Information about the creator of a recipe, the creation time or the location may be important for selecting a good recipe: If a very similar robot created it, the likelihood that it will work will be higher. The older a map is, the more likely it is outdated.

Related language constructs:

```
creationDateTime(Thing, xsd:dateTime)
createdBy(Thing, Thing)
```

3.1.2 Performance data

The number of times a recipe has been downloaded, the amount of successful executions, or the kinds of robots it has been tried on are valuable pieces of information when trying to select one recipe among several alternatives. So far, the language only has basic support for this kind of information, but once we gather experience which information is actually relevant, we will extend these capabilities.

Related language constructs:

```
hasNumOfAttempts(PurposefulAction, xsd:positiveInteger)
hasNumOfSuccesses(PurposefulAction, xsd:positiveInteger)
hasSuccessProbability(PurposefulAction, xsd:double)
```

3.1.3 Units and coordinate systems

When using numeric data, it is important to know what these numbers denote, e.g. which units are used and which coordinate system spatial information is described in. Without such information explicitly represented, all the recipe interpretation would be based only on informal conventions. The respective language constructs will be extended once the requirements of the other work packages in terms of information that needs to be represented have become clearer.

Related language constructs:

```
CoordinateSystem
CartesianCoordinateSystem
correspondingCoordinateSystem(PhysicalSituation, CoordinateSystem)
```

3.2 Tasks and Actions

This section describes how actions and tasks are modeled in the RoboEarth language. There are two kinds of descriptions: recipes, i.e. general task descriptions that can be instantiated by executing them, and models of concrete, i.e. observed actions, which can for instance be used in log files. Remember that Description Logic, the formalism underlying the RoboEarth language, distinguishes between terminological descriptions (TBOX) and assertional knowledge (ABOX). The former describes classes, their relations and properties, while the latter contains instances of these classes. In case of actions, these instances may be

- actually observed or performed actions (something that happened at a certain time)
- planned actions (something the robot intends to do)
- inferred actions (something the robot imagines that happens)
- asserted actions (some action someone told the robot has happened)

In contrast, the TBOX describes general information about classes of actions. These can be quite general classes like *PuttingSomethingSomewhere*, or very specific ones like *PuttingDinnerPlateInCenterOfPlacemat*. However, all these class specifications describe types of actions that, when they are actually executed, get instantiated to the corresponding actions in the ABOX.

3.2.1 Action classes

The RoboEarth ontology provides a taxonomic structure describing several subclasses of Action:

```
* Action
  * PurposefulAction
    * Perceiving (similar to SensoryEvent, but in the action context)
    * VoluntaryBodyMovement, e.g. Reaching or ReleasingGrasp (Movements that are
        not directly manipulating an object)
    * ActionOnObject (any kind of object interaction)
      * ControllingSomething tap, electrical device, ...
      * HoldingAnObject (different grasps, no movement involved)
      * Movement-Translation picking up, putting down, or moving objects, walking, ...
      * RemovingSomething cleaning activities
      * OpeningSomething bottle, cupboard, drawer, ...
      * ClosingSomething bottle, cupboard, drawer, ...
```

There are many more and more detailed action classes available, and the taxonomy can easily be extended. These action classes form the vocabulary for describing the different kinds of actions the system knows. An overview of most existing action classes can be found in the Appendix.

Each of these actions can be described by its properties: For instance, one could state that every action of type *Movement-TranslationEvent* is a subclass of Action with the properties *fromLocation*, and *toLocation* set. This description can be used to sort any instance of an action that has these properties into the class *Movement-TranslationEvent*.

As an example, let's have a look at the description of the class *Putting-SomethingSomewhere* that corresponds to transporting an object from one position to another. Obviously, this kind of action involves to pick up the object, move to the goal position, and put the object down again. These sub-actions are modeled in the following piece of code:

```
<owl:Class rdf:about="#PuttingSomethingSomewhere">

    <rdfs:subClassOf rdf:resource="#Movement-TranslationEvent"/>
    <rdfs:subClassOf rdf:resource="#TransportationEvent"/>

    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#subAction"/>
            <owl:someValuesFrom rdf:resource="#PickingUpAnObject"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#subAction"/>
            <owl:someValuesFrom rdf:resource="#CarryingWhileLocomoting"/>
        </owl:Restriction>
    </rdfs:subClassOf>

    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#subAction"/>
            <owl:someValuesFrom rdf:resource="#PuttingDownAnObject"/>
        </owl:Restriction>
    </rdfs:subClassOf>

    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#orderingConstraints"/>
            <owl:hasValue rdf:resource="#SubEventOrderingPuttingSomethingSomewhere1"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#orderingConstraints"/>
            <owl:hasValue rdf:resource="#SubEventOrderingPuttingSomethingSomewhere2"/>
        </owl:Restriction>
    </rdfs:subClassOf>

</owl:Class>
```

Note that the ordering of the three *subAction* restrictions in the OWL file does not have any meaning. In order to describe their allowed orderings, we need to add some constraints. These constraints are described as follows. Unfortunately, the constraints cannot be checked automatically by a description logic reasoner, which is unable to resolve the relations among sub-actions of a task due to limitations in the DL languages, but applications need to verify themselves if they hold.

```
<PartialOrdering-Strict rdf:about="#SubEventOrderingPuttingSomethingSomewhere1">
    <occursBeforeInOrdering rdf:resource="#PickingUpAnObject"/>
    <occursAfterInOrdering rdf:resource="#CarryingWhileLocomoting"/>
</PartialOrdering-Strict>

<PartialOrdering-Strict rdf:about="#SubEventOrderingPuttingSomethingSomewhere2">
    <occursBeforeInOrdering rdf:resource="#CarryingWhileLocomoting"/>
    <occursAfterInOrdering rdf:resource="#PuttingDownAnObject"/>
</PartialOrdering-Strict>
```

3.2.2 Composing actions to tasks

Complex robot tasks can be decomposed into primitive actions and movements. If the sub-actions for lower-level actions are already modeled, tasks can be described conveniently on a rather abstract level, like the already mentioned *PuttingSomethingSomewhere* actions. The following code is an excerpt of a plan for setting a table. The upper region describes the task *SetATable* as a subclass of *Action* with a set of *subAction*s.

```
<owl:Class rdf:about="#SetATable">
    <rdfs:subClassOf rdf:resource="&knowrob;Action"/>
    <rdfs:label rdf:datatype="&xsd;string">set a table</rdfs:label>
    <owl:equivalentClass>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
                <owl:Restriction>
                    <owl:onProperty rdf:resource="&knowrob;subEvents"/>
                    <owl:someValuesFrom rdf:resource="#PuttingSomethingSomewhere1"/>
                </owl:Restriction>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="&knowrob;subEvents"/>
                    <owl:someValuesFrom rdf:resource="#PuttingSomethingSomewhere2"/>
                </owl:Restriction>

            [...]

            </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="#PuttingSomethingSomewhere1">
    <owl:equivalentClass>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
                <owl:Restriction>
                    <owl:onProperty rdf:resource="&knowrob;objectActedOn"/>
                    <owl:someValuesFrom rdf:resource="&knowrob;PlaceMat"/>
                </owl:Restriction>
                <owl:Class rdf:about="&knowrob;PuttingSomethingSomewhere"/>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="&knowrob;toLocation"/>
                    <owl:someValuesFrom rdf:resource="#Place1"/>
                </owl:Restriction>
            </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="#Place1">
    <owl:equivalentClass>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
                <owl:Restriction>
                    <owl:onProperty rdf:resource="&knowrob;inFrontOf-Generally"/>
                    <owl:someValuesFrom rdf:resource="&knowrob;Chair-PieceOfFurniture"/>
                </owl:Restriction>
                <owl:Class rdf:about="&knowrob;Place"/>
            </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>

[...]
```

The lower part shows how task-specific subclasses describe actions with certain parameters. While setting a table, the robot may be told to put a place mat in front of the chair. This command is translated into a class specification: a subclass of *PuttingSomethingSomewhere* actions with a *PlaceMat* as *objectActedOn* and a *toLocation Place1*, which by itself is described as some *Place* which is *inFrontOf-Generally* of some *Chair-PieceOfFurniture*. The same mechanism can be used to describe other action parameters like grasp types or trajectories.

The modeling of such tasks in the TBOX, i.e. using restrictions on the properties of classes, may seem a bit clumsy at first glance, especially compared to an ABOX model in which the properties of instances can directly be asserted. However, it is necessary to describe action recipes on the TBOX level since they are template-like action descriptions that are instantiated by executing them. Even though the classes may be small (There may not be a lot of actions where a place mat is positioned in front of a chair), they are still classes of actions that comprise sets of action instances.

Figure 3.1 visualizes an action recipe that only lists some navigation actions to perform, namely two *LinearVelocityCommands* and two *AngularVelocityCommands*, including their ordering relations. This very simple recipe was used for the demonstrator planned for the first RoboEarth workshop. A robot was tele-operated by a human and created such a recipe describing the performed actions. It uploaded the recipe to the RoboEarth knowledge base, another robot downloaded the recipe and performed the actions.

Related language constructs:

```
orderingConstraints(Thing, MathematicalOrdering)
```

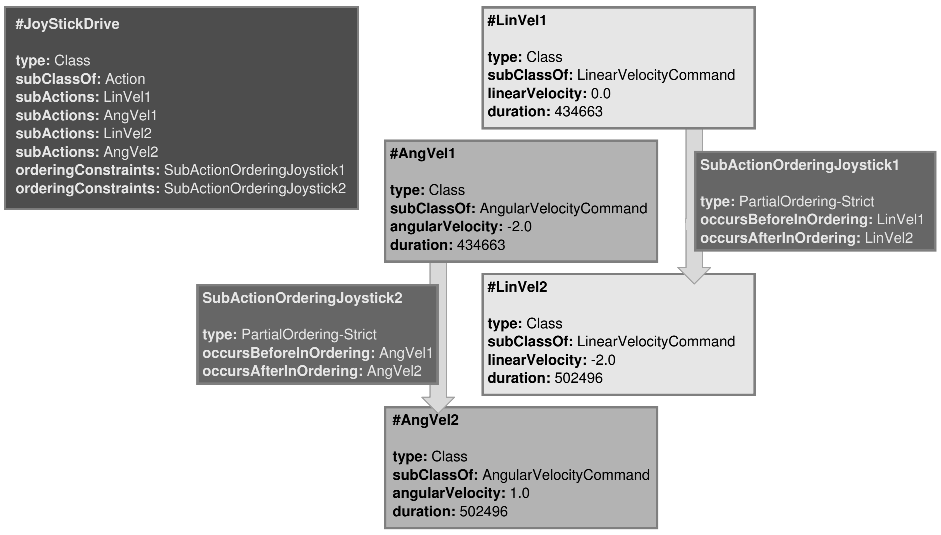

#JoyStickDrive
type: Class
subClassOf: Action
subActions: LinVel1
subActions: AngVel1
subActions: LinVel2
subActions: AngVel2
orderingConstraints: SubActionOrderingJoystick1
orderingConstraints: SubActionOrderingJoystick2
#LinVel1
type: Class
subClassOf: LinearVelocityCommand
linearVelocity: 0.0
duration: 434663
#AngVel1
type: Class
subClassOf: AngularVelocityCommand
angularVelocity: -2.0
duration: 434663
SubActionOrderingJoystick1
type: PartialOrdering-Strict
occursBeforeInOrdering: LinVel1
occursAfterInOrdering: LinVel2
SubActionOrderingJoystick2
type: PartialOrdering-Strict
occursBeforeInOrdering: AngVel1
occursAfterInOrdering: AngVel2
#LinVel2
type: Class
subClassOf: LinearVelocityCommand
linearVelocity: -2.0
duration: 502496
#AngVel2
type: Class
subClassOf: AngularVelocityCommand
angularVelocity: 1.0
duration: 502496

Figure 3.1: Example action recipe. The block in the upper left defines the action recipe including the sub-actions and ordering constraints. The blocks on the right side visualize the four actions involved in the task and the partial order imposed by the two ordering constraints.

```
subAction(Action, Action)
objectActedOn(Action, EnduringThing-Localized)
fromLocation(Movement-Translation, EnduringThing-Localized)
toLocation(Movement-Translation, EnduringThing-Localized)
primaryObjectMoving(ActionOnObject, EnduringThing-Localized)

after(TimePoint, TimePoint)
duration(TemporalThing, TimeInterval)
startTime(TemporallyExtendedThing, TimePoint)
endTime(TemporallyExtendedThing, TimePoint)
temporallySubsumes(TemporalThing, TemporalThing)

doneBy(Action, Agent-Generic)
bodyPartUsed(VoluntaryBodyMovement, AnimalBodyPart)

postureDuringMovement(Action, Posture-Configuration)
trajectory-Complete(Action, Trajectory)
trajectory-Arm(Action, ArmTrajectory)
```

3.2.3 Describing observed actions

Whenever the robot is reasoning on actually performed actions, either by itself or by a human, it needs to describe action instances. These instances can, for example, be generated by an action recognition system that interacts with the knowledge base and populates the set of action instances based on observations of humans. Based on these observations, the system can set parameters like the *startTime*, the *objectActedOn*, or the *bodyPartUsed*. For more information, see [6] or [4].

3.3 Objects and Object Recognition Models

Three kinds of object-related information can be exchanged: Information about the properties of object classes, descriptions how objects of types can be detected by the robot, and information about concrete object instances that were detected in an environment. Figure 3.2 illustrates the relation between the different pieces of information: The bottom block describes the object recognition model, the upper three blocks describe one object of type *Cup* that has been perceived in the environment. More details on these descriptions can be found in the next sections.

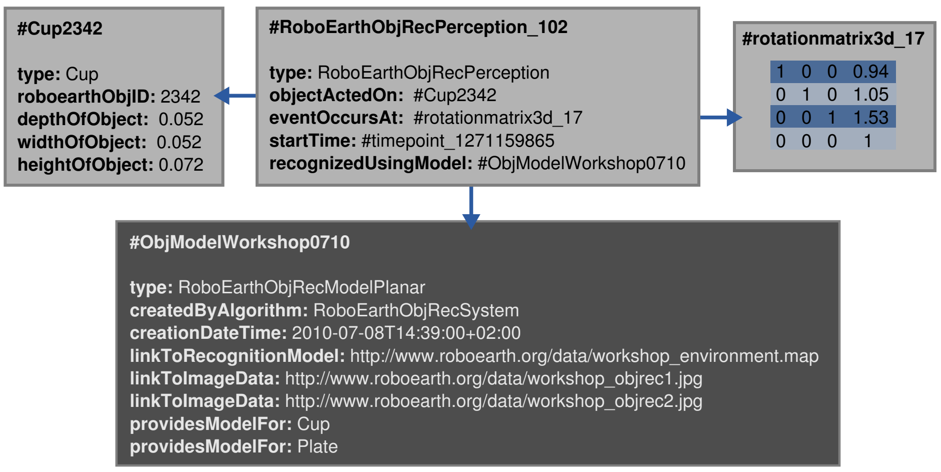

#Cup2342
type: Cup
roboearthObjID: 2342
depthOfObject: 0.052
widthOfObject: 0.052
heightOfObject: 0.072
#RoboEarthObjRecPerception_102
type: RoboEarthObjRecPerception
objectActedOn: #Cup2342
eventOccursAt: #rotationmatrix3d_17
startTime: #timepoint_1271159865
recognizedUsingModel: #ObjModelWorkshop0710
#rotationmatrix3d_17
1 0 0 0.94
0 1 0 1.05
0 0 1 1.53
0 0 0 1
#ObjModelWorkshop0710
type: RoboEarthObjRecModelPlanar
createdByAlgorithm: RoboEarthObjRecSystem
creationDateTime: 2010-07-08T14:39:00+02:00
linkToRecognitionModel: http://www.roboearth.org/data/workshop_environment.map
linkToImageData: http://www.roboearth.org/data/workshop_objrec1.jpg
linkToImageData: http://www.roboearth.org/data/workshop_objrec2.jpg
providesModelFor: Cup
providesModelFor: Plate

Figure 3.2: Description of an object recognition model (lower block) and the detection of an object using that model (upper blocks).

3.3.1 Object classes

All object types are organized in a taxonomic structure, from very general classes like *SpatialThing* to specific ones like *Refrigerator-Freezer*. A picture of this taxonomy, containing most of the object classes currently present in the RoboEarth language, can be found in the Appendix. The taxonomic structure has the advantage that knowledge can be represented at different levels of granularity: For instance, one could state that a *Container* can *contain SpatialThing*s. Such a property will be inherited by all subclasses and would need to be asserted only once.

Multiple inheritance reflects the different aspects of a class: For instance, a *MicrowaveOven* is a *FoodOrDrinkPreparationDevice*, a kind of *Oven*, and also an *ElectricalHouseholdAppliance*, inheriting all properties of the respective class trees. This multi-faceted modeling is very important to capture the complexity of real-world environments.

Classes are not only arranged in the taxonomic structure, but futher described by properties, e.g. that the primary function of an *Oven* is *HeatingFood*, and that it has a *Handle* as *properPhysicalPart*. If an object possesses all required properties, it can be automatically classified as an instance of the respective class. Properties are also arranged in a hierarchical structure, as for example in the following list of language constructs (*spatiallyRelated* is the super-property of *connectedTo* or *topologicalRelations*).

Related language constructs:

```
contains
mainColorOfObject
objectShapeType
typePrimaryFunction-StoragePlaceFor

spatiallyRelated
  connectedTo
    rotationallyConnectedTo
      hingedTo
  directionalRelations
    aboveOf
    behind-Generally
    belowOf
    inFrontOf-Generally
    toTheSideOf
      toTheLeftOf
      toTheRightOf
  topologicalRelations
    in-ContGeneric
    on-Physical
    inCenterOf
    outsideOf
```

3.3.2 Object recognition models

Having only an OWL model that describes an object class, a robot is, in general, not able to recognize this object. For this purpose, it needs more detailed information that describes the object's shape, texture, appearance, or salient feature points and that can be used to parameterize an object recognition system. These recognition models are, on the one hand, required to perform an action, and are, on the other hand, something robots can exchange.

A recognition model has to be linked to both the object classes it allows to recognize and to the recognition system that can load and use this model. This information can be used by a robot to find out if it has the prerequisites for using this model.

Since the recognition models can be quite large and complex, and since the central RoboEarth knowledge base does not need to perform reasoning on the details described in the model (for instance, some image feature descriptors that do not serve any other purpose than recognition), we decided not to convert the models completely into an OWL based format, but to store them in the object recognition system's binary format. This binary file is then annotated in the OWL-based RoboEarth language so that automated reasoning on the provided object classes, the prerequisites becomes possible.

Related language constructs:

```
createByAlgorithm
linkToRecognitionModel
linkToImageData

providesModelFor
recognizedUsingModel
```

3.3.3 Object instances

When a robot has explored an environment, it may want to exchange information about the locations of objects it found. Other robots could profit from that information to quickly find the respective items. These are concrete physical objects and therefore have to be described at the instance level (ABOX).

A naive approach would be to add a property *location* that links an object instance to a point in space or, more general, the homography pose matrix. However, this approach is limited to describe either the current state of the world or a static environment – one cannot express that the object states and locations change over time. This is a strong limitation since robots need to be able to describe past and (predicted) future states as well as hypothetical effects of actions.

Such statements are not possible with this naive approach since OWL only supports binary relations between exactly two entities. These relations can only express if something is related or not, or if it has a property or not. They cannot qualify these statements by saying that a relation held an hour ago, or is supposed to hold with a certain probability. For this purpose, we need an additional instance in between that links e.g. the object, the location, the time, and the probability.

In KnowRob, and thus also in the RoboEarth language, these links are represented by the event that established them: the perception of an object, an inference process, or the prediction of future states based on projection or simulation (Figure 3.3. There can, of course, be multiple events assigned to one object, corresponding to different detections over time. This representation allows to describe states that change over time, to perform reasoning on the source of information, and handle contradicting statements, e.g. from different perception modules. In the context of RoboEarth, such conflicts could arise when the robot downloaded information that an object is at a certain place, though it cannot find it there, but at a different location.

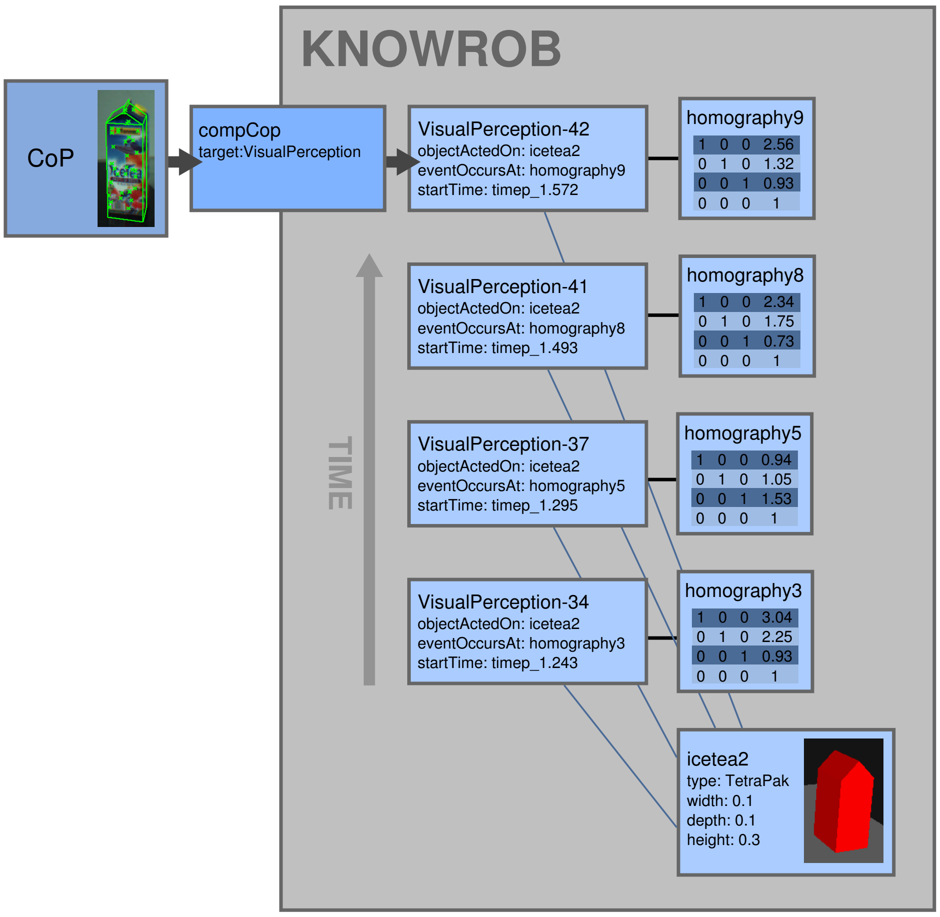


Figure 3.3: Visualization of the internal object representation. Based on information from the vision system, KnowRob generates *VisualPerception* instances that link the object instance *icetea2* to the different locations where it is detected over time.

All perceptions or inference results are represented as subclasses of *MentalEvent*, for instance *VisualPerception* or *Reasoning*. Most object recognition algorithms will be described as sub-classes in the *VisualPerception* tree. KnowRob also provides predicates (holds(rel, T) and holds_tt(rel, [St, End])) that operate on this representation and compute if a relation is true at a given point in time.

Related language constructs:

```
Remembering
Reasoning
ThoughtExperimenting
Perceiving
  TouchPerception
  RFIDPerception
  VisualPerception
    SiftMatching
    ShapeModelMatching

eventOccursAt
objectActedOn
startTime
```

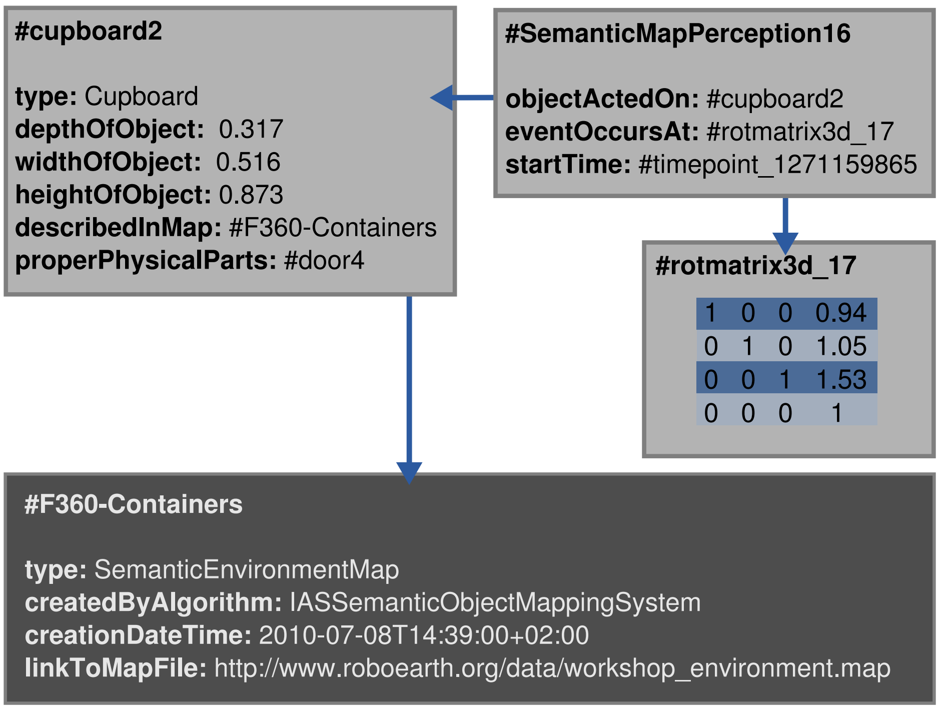
3.4 Environment Maps

Environment information has been described in various ways in robotics, just to name a few:

- Occupancy grid maps describe obstacles and free space in a grid-based structure.
- Topological maps describe the environment as a graph in which the vertices correspond to points of interest and the edges mean that one vertex can be reached from an adjacent one.
- Point cloud maps describe the surface of the environment by a set of points in 3D space.
- Object maps consist of localized, typed objects that have been recognized in the environment.
- Maps of visual landmarks contain (often only visually distinctive, but otherwise meaningless) points in space that can serve for localization.

With the RoboEarth language, we support all of these kinds of maps by providing two ways to describe maps: The map can either be described as a set of instances inside the OWL file, or as a link to an external (binary) file.

The first approach is especially suited for those maps where logical reasoning over the elements in the map is desired, may they be detected objects, walls, doors or similar things. The second approach has advantages if the maps are large and if no logical reasoning is required (e.g. point cloud or voxel-based maps).



#cupboard2
type: Cupboard
depthOfObject: 0.317
widthOfObject: 0.516
heightOfObject: 0.873
describedInMap: #F360-Containers
properPhysicalParts: #door4
#SemanticMapPerception16
objectActedOn: #cupboard2
eventOccursAt: #rotmatrix3d_17
startTime: #timepoint_1271159865
#rotmatrix3d_17
1 0 0 0.94
0 1 0 1.05
0 0 1 1.53
0 0 0 1
#F360-Containers
type: SemanticEnvironmentMap
createdByAlgorithm: IASSemanticObjectMappingSystem
creationDateTime: 2010-07-08T14:39:00+02:00
linkToMapFile: http://www.roboearth.org/data/workshop_environment.map

Figure 3.4: Encoding of an environment map that combines a binary file (linked using the *linkToMapFile* property) with an object that was recognized in the respective environment.

Both approaches can be combined: For example, an occupancy grid map that serves for describing free space, localization and path planning, can be combined with a set of objects and their positions in the environment, as illustrated in Figure 3.4. The detected objects are represented by the detection process as explained in Chapter 3.3.

In both cases, there is an owl file that describes the semantics of the provided data, even if the map itself is stored as a binary file. This way, the power of the RoboEarth language in terms of matching and reasoning can also be applied to maps that are stored as separate files.

Related language constructs:

`describedInMap`

```
linkToMapFile
createdByAlgorithm

MetricMap
  ObjectMap
  OccupancyGridMap
  PointMap
  LandmarkMap

TopologicalMap
  LandmarkMap
  ObjectMap

TwoDimensionalMap
ThreeDimensionalMap
```

Chapter 4

Reasoning in the RoboEarth Language

Since the RoboEarth language is implemented as a formal, logical language, it allows for automated reasoning. The following sections give some example applications that the recipes can be used for. These examples are meant to give an intuition of what is possible with this language. They are currently not implemented, but in some of the cases, very similar problems have been solved with the KnowRob language (of which the RoboEarth language is only an extension). In these cases, we provide a reference to a paper with more detailed information.

4.1 Determining and retrieving missing information

For executing a recipe, the robot may need additional models and parameterizations like an environment map to navigate or object models to recognize the objects that are mentioned in the recipe. It may also need additional action recipes that provide a more fine-grained description of actions that are only contained as atomic entities in a high-level recipe.

Using the RoboEarth language, the robot can recursively retrieve missing components while flexibly matching what it has and what is needed, exploiting sub-class and super-class relations.

4.2 Generating plans

The RoboEarth action recipes are not executable code, they need to be interpreted in order to perform the described tasks. However, the abstract task model can be used to generate code for the robot's planning system. This process comprises the following tasks:

- The abstract action descriptions need to be transformed into parameterizations of executable action routines.
- Descriptions of objects and locations have to be related to the robot's percepts (often referred to as the "grounding" problem).
- The planning system has to make sure that constraints defined in the recipe are satisfied in the plan execution (e.g. a partial order or conditional execution).

Most of these tasks are anything but trivial, [3] discusses some of the challenges and presents approaches to solve them. In RoboEarth, the execution component developed as part of WP3 Labeling will be in charge of approaching these issues.

4.3 Verifying action execution

The abstract action descriptions can not only be used both for generating behavior [8], but also for recognizing actions [4]. Given a knowledge processing system that is running on the robot and grounded in its perception and action system, the recipe declarations can be used to verify if an action has been performed correctly, and if not, which of the sub-actions failed [5]. KnowRob, for instance, can provide much of the needed information.

4.4 Generating more abstract or more specific representations

The RoboEarth ontology describes what sub-actions a task is composed of – what can be used to either abstract a sequence of actions, that is described with fine granularity, into a coarser-grained representation, or to transform a coarse description into a detailed action sequence. One important application would be to upload rather high-level task descriptions instead of very detailed (and thus more likely hardware-dependent) ones. The other direction is needed when executing a recipe when the robot has to determine a sequence of low-level actions based on an abstract, high-level specification.

In [4], the authors demonstrated the automated generation of more abstract action descriptions on observed human actions described in the KnowRob language. Human actions are even more complex to handle than most robot actions, since the only available information is obtained from observation. When recording action recipes, a robot does not have to rely on external observation, but already knows what task it is performing, and can use this information to better structure the recipe.

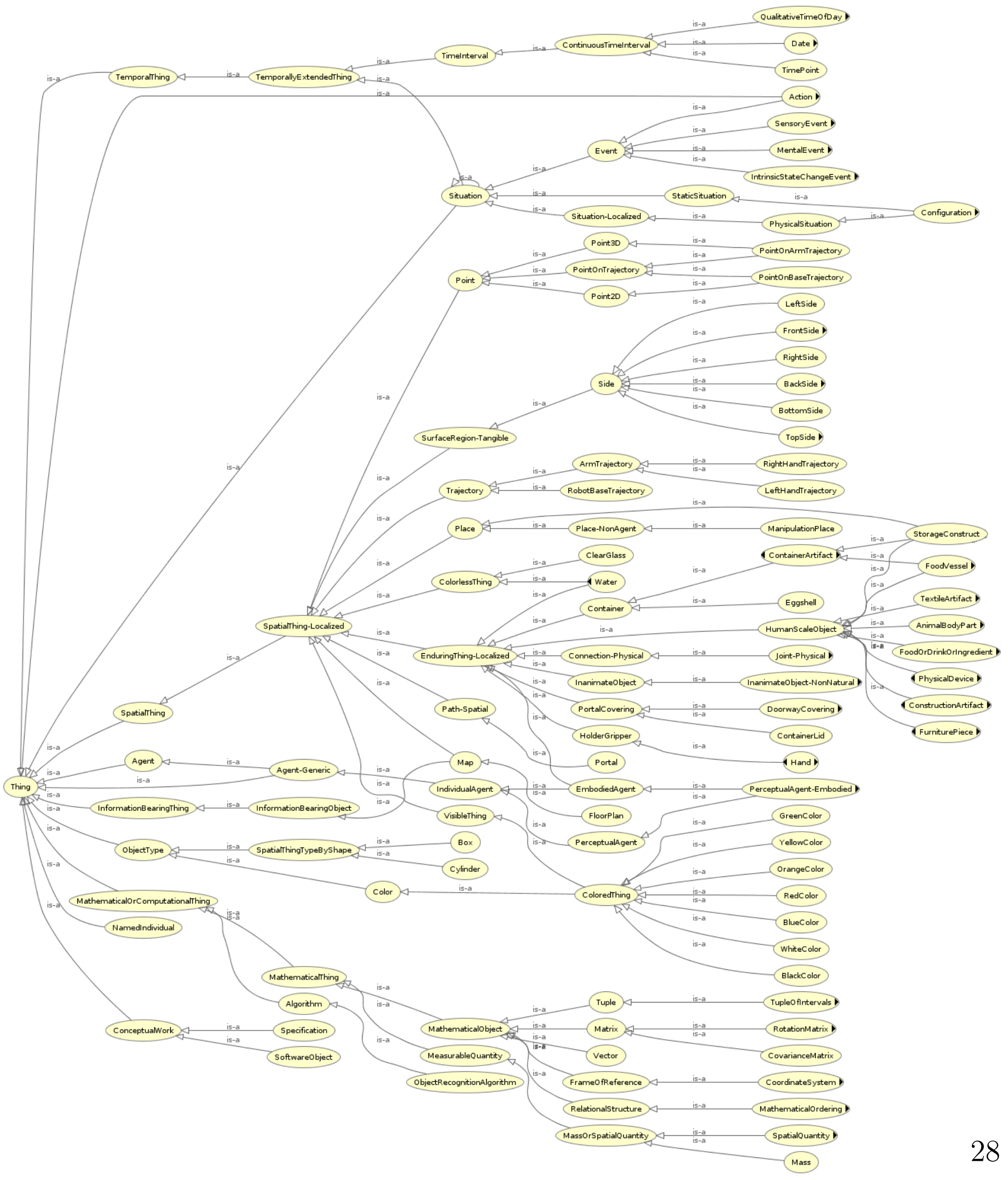
4.5 Merging recipes

If different robots upload action descriptions, there will probably be redundant recipes: For instance, one robot may describe an action as “reach towards the bottle, grasp the bottle, pick it up, move the gripper to a carrying pose, move the base to a pose from which the desired position of the bottle can be reached, put the bottle at that location, release the grasp, and retract your hand”. The same task can be described as “pick up a bottle, move to the destination, put down the bottle” or “transport the bottle to location A”.

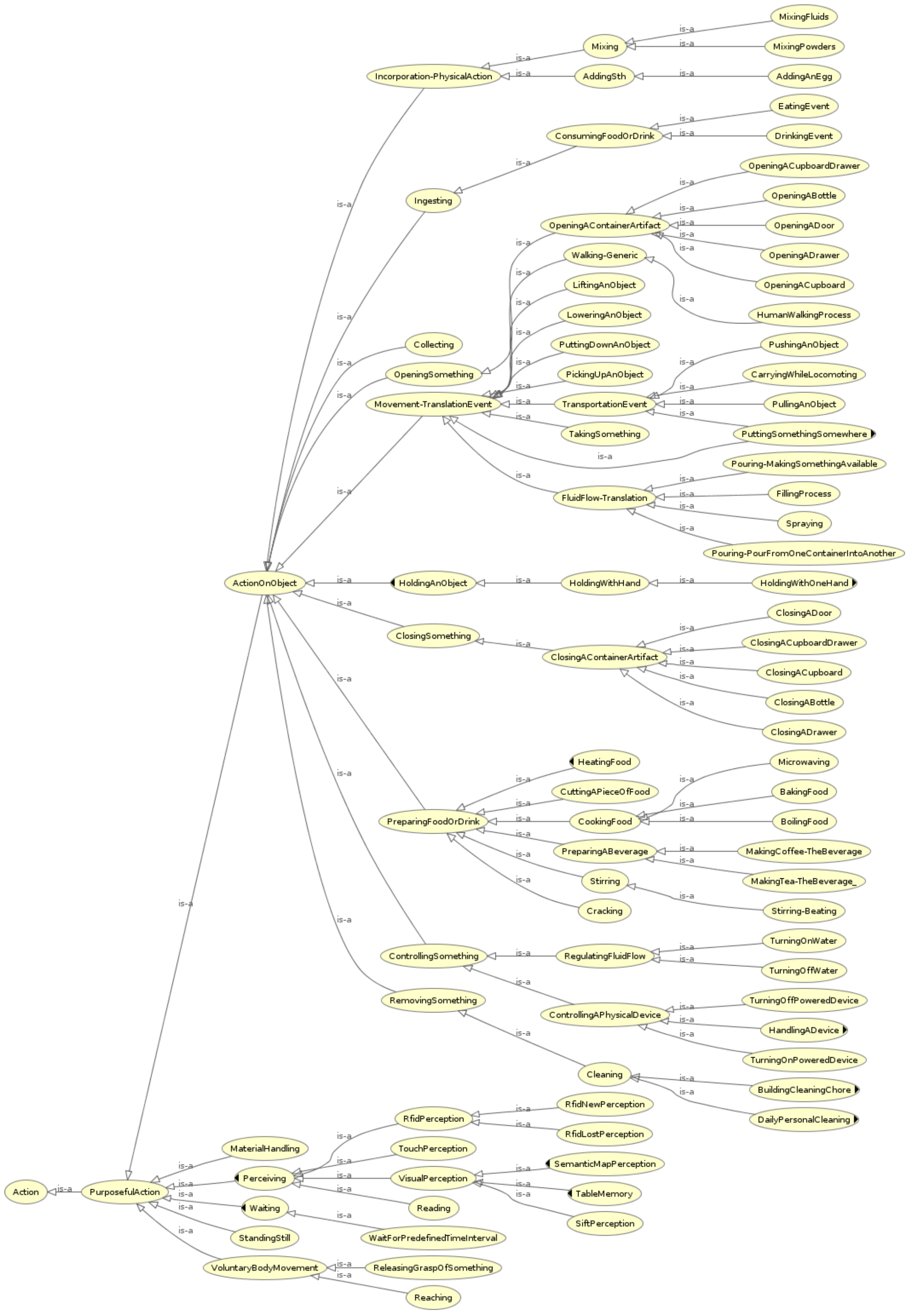
Using the action descriptions in the ontology and logical inference, the system can detect that these descriptions describe the same task, though at different levels of abstraction, and fuse them to one recipe. In addition to this temporal (de)composition of actions, the system can also handle descriptions at different levels of the class hierarchy: One robot may describe an action as *OpeningSomething*, another one as the more specific *OpeningAContainerArtifact*, and a third one talk about *OpeningABottle*.

Appendix: Ontology overview

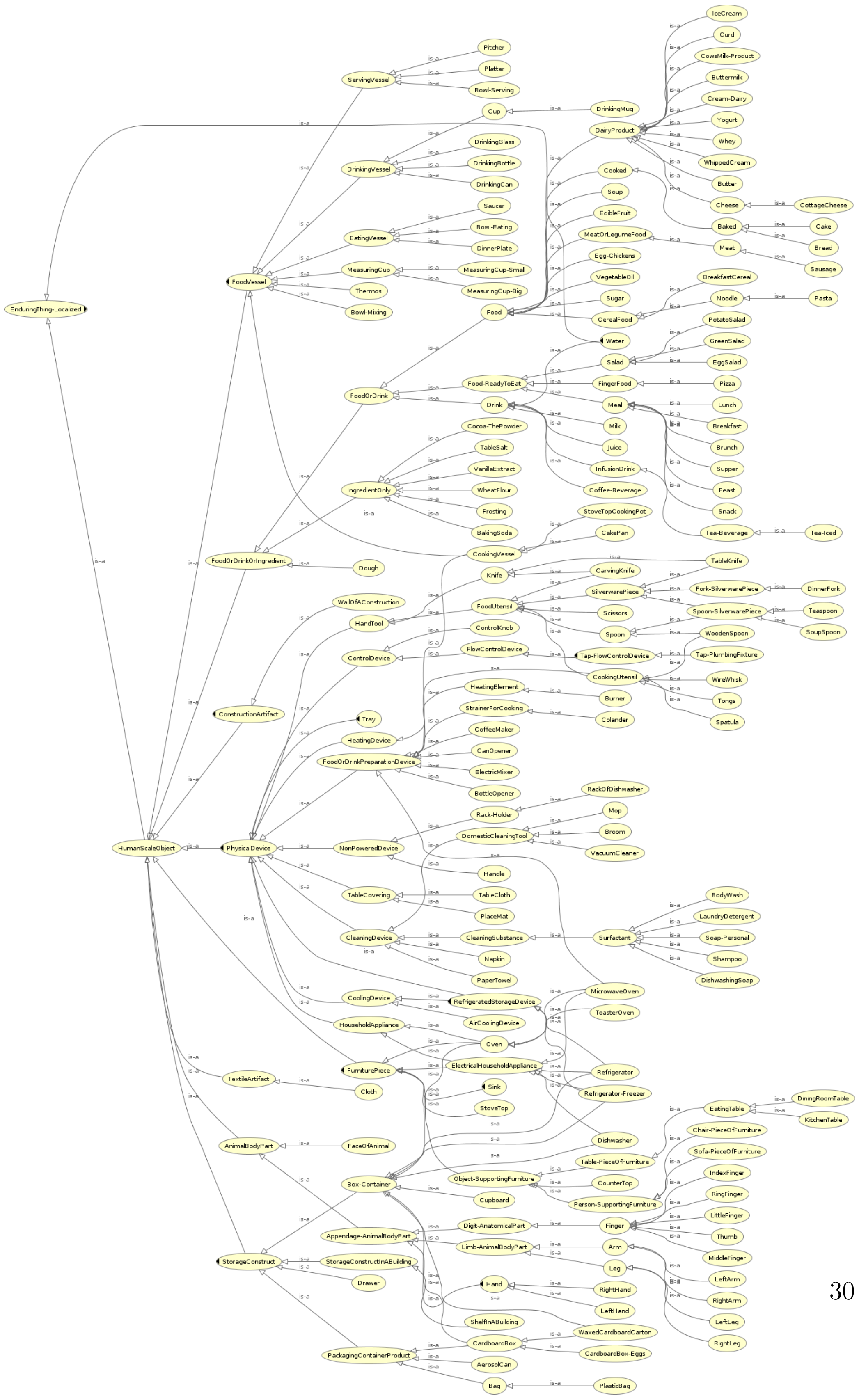
Upper ontology



Action ontology



Object ontology



Bibliography

[1] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider. *The description logic handbook*. Cambridge University Press New York, NY, USA, 2007.

[2] Franz Baader, Ian Horrocks, and Ulrike Sattler. Chapter 3 description logics. In Vladimir Lifschitz Frank van Harmelen and Bruce Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 135–179. Elsevier, 2008.

[3] Michael Beetz, Dominik Jain, Lorenz Mösenlechner, and Moritz Tenorth. Towards Performing Everyday Manipulation Activities. *Robotics and Autonomous Systems*, 2010. To appear.

[4] Michael Beetz, Moritz Tenorth, Dominik Jain, and Jan Bandouch. Towards Automated Models of Activities of Daily Life. *Technology and Disability*, 22, 2010.

[5] Lorenz Mösenlechner, Nikolaus Demmel, and Michael Beetz. Becoming Action-aware through Reasoning about Logged Plan Execution Traces. In *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, 2010. accepted for publication.

[6] Moritz Tenorth, Jan Bandouch, and Michael Beetz. The TUM Kitchen Data Set of Everyday Manipulation Activities for Motion Tracking and Action Recognition. In *IEEE Int. Workshop on Tracking Humans for the Evaluation of their Motion in Image Sequences (THEMIS). In conjunction with ICCV2009*, 2009.

[7] Moritz Tenorth and Michael Beetz. KnowRob — Knowledge Processing for Autonomous Personal Robots. In *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, 2009.

[8] Moritz Tenorth, Daniel Nyga, and Michael Beetz. Understanding and Executing Instructions for Everyday Manipulation Tasks from the World

Wide Web. In *IEEE International Conference on Robotics and Automation (ICRA).*, 2010.