Yoda Easy Guide

David Cohen
ECE Department – Carnegie Mellon University

Introduction

YODA, Yet another Ontological Dialog Architecture is a sophisticated tool to allow developers to quickly build intelligent spoken dialog systems. This tutorial will introduce the major components and capabilities of Yoda by walking through the creation of a simple messaging calendar assistant. We envision this system as a smartphone app, which the user can use to message their friends and coordinate meetings.

Understand Your Dialog System's Desired Functionality

The YODA dialog manager is designed to build dialog systems which combine the functions of information retrieval, and information entry dialog systems. An IR dialog system's main purpose is to answer questions about a database of objects - such as restaurants, movies, or events in a calendar. An information entry system allows the user to create descriptions of objects for entry in a database - such as scheduling events in a calendar. More complex systems can be built on top of these two basic functions. For example, a transactional dialog system, which may take actions on behalf of the user such as making reservations or sending emails, is just an information entry system with additional functionality after the appropriate information has been entered.

The first step for a YODA system developer is to decide what databases it is intended to access and in what way. This involves enumerating the types of things the system can talk about, and how it can talk about them. We'll do this informally at first, then more formally in section

Our demonstration system can talk about people, meetings, times, and emails. Every meeting has a time and several attendees. Every email has a sender, a recipient, and some text content. We want the system to be able to set up a meeting, as well as answer basic queries about meetings. We also want the system to be able to answer basic queries about emails, and listen to and send dictated emails.

Build the Ontology

A YODA developer must then formalize the objects, actions and properties which are relevant to their domain by extending the YODA skeleton ontology with new classes, properties, values, and instances. Objects should be given appropriate slots, relations, and properties which correspond to the actual objects. There are several powerfull tools available for editing OWL ontologies, but we recommend protege, and all our examples are shown in protege.

Figure 1 shows the class hierarchy for the domain ontology.

Define the Dialog Functionality

Dialog Tasks

The developer defines what dialog tasks

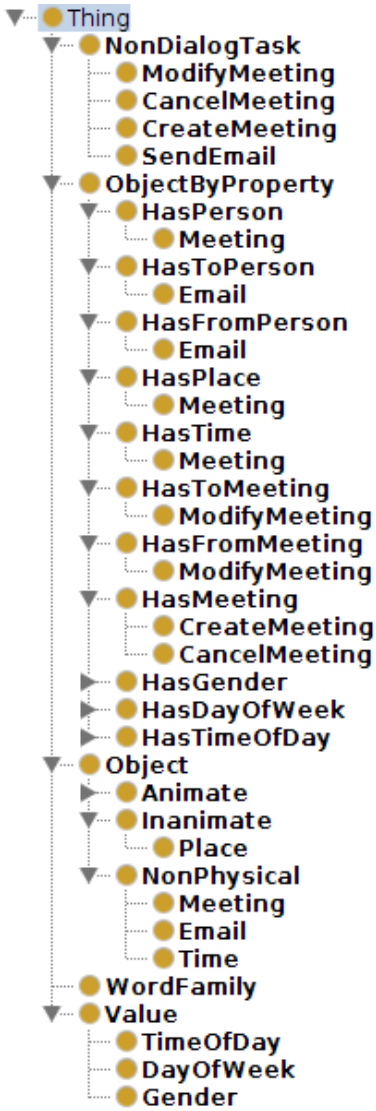

Figure 1: YODA sample domain ontology

Non-dialog Tasks

Several aspects of a non-dialog task will effect how the dialog manager interacts with them. In addition to implementing the execution of these tasks, the developer must define the information relevant for dialog management.

Figure 2 shows the definition for the NonDialogTaskPreferences class. It contains several parameters which determine the reward and penalty (negative reward) received by the system for executing a task, delaying execution of a task, and incorrectly executing a task or executing a task with incorrect parameters. These preferences are used by the dialog manager to perform decision-theoretic dialog management, and may be used to train reinforcement-learning dialog managers in the future.

There is another parameter, alwaysRequireConfirmation, that can be set to true if the dialog manager is required to get a confirmation from the user before performing an non-dialog task. Depending on the cost / benefit tradeoff, the dialog manager may decide to request a confirmation before executing a non-dialog task even if the flag is not set, and setting a high penalty for incorrect execution can make explicit confirmation requests more likely, but setting the flag allows the designer to enforce strict transparency.

```
public class NonDialogTaskPreferences {
    // if explicit confirmation is always required before the
    // task is executed, set alwaysRequireConfirmation to true
    public boolean alwaysRequireConfirmation;

    // all the rewards and penalties are given as positive
    // numbers, penalties are converted to negative numbers for
    // decision-making
    public double penaltyForDelay;
    public double rewardForCorrectExecution;
    public double penaltyForIncorrectExecution;

    // define the slots that are required to be defined by the
    // user before execution is possible
    public Set<String> slotsRequired;

    public NonDialogTaskPreferences(
        boolean alwaysRequireConfirmation,
        double penaltyForDelay,
        double rewardForCorrectExecution,
        double penaltyForIncorrectExecution,
        Set<String> slotsRequired) {
        this.alwaysRequireConfirmation =
            alwaysRequireConfirmation;
        this.penaltyForDelay = penaltyForDelay;
        this.rewardForCorrectExecution =
            rewardForCorrectExecution;
        this.penaltyForIncorrectExecution =
            penaltyForIncorrectExecution;
        this.slotsRequired = slotsRequired;
    }
}
```

Figure 2: The NonDialogTask interface

In our Figure 4 line 5-7, we show the preferences declaration for the send-email task. Here, we set alwaysRequireConfirmation to true, so the user will always be asked for explicit confirmation before an email is sent, as in the following example:

User Send an email to Jerry.
System You want me to send an email to Jerry, right?
User Yes.
System Ok.
System <executes task>

In our Figure 5 line 5-10, we show the preferences declaration for the create-meeting task. Here, we set alwaysRequireConfirmation to false, so the system is allowed to perform the task without an explicit confirmation, as in the following example:

User Set up a meeting this afternoon with Jerry in my office.
System Ok.
System <executes task>

The last parameter of the NonDialogTaskPreferences class sets requirements for what must be specified for a task to be executable. As is shown in the preference declarations from Figures 4 and 5, sending an email only requires a sender, while creating a

meeting requires a description of a meeting with at least one time, place, and attendee. The dialog manager will not consider executing a task until all its required slots are filled, and will instead engage in slot-filling dialog, as in the following example:

```
User Set up a meeting this afternoon.
System Where will this meeting be?
User In my office.
System Who is this meeting with?
User Jerry.
System Ok.
System <executes task>
```

After basic preferences have been defined, the NonDialogTask interface, shown in figure 3, must be implemented for each non-dialog task. Two implementations of this are shown in this tutorial, SendEmailTask is shown in figure 4, and CreateMeetingTask is shown in figure 5

```
1  public interface NonDialogTask {
2
3      public enum TaskStatus {SUCCESSFULLY_COMPLETED,
4        CURRENTLY_EXECUTING_BLOCKING,
5        CURRENTLY_EXECUTING_NOT_BLOCKING, FAILED}
6
7      // return preferences object
8      public NonDialogTaskPreferences getPreferences();
9
10     // the probability that the taskSpec can be executed (0-1)
11     public double assessExecutability(SemanticsModel taskSpec);
12
13     // return the string ID of the executing task (taskID)
14     public String execute(SemanticsModel taskSpec);
15
16     // return the string status indicator for the taskID
17     public TaskStatus status(String taskID);
18
19 }
```

Figure 3: The NonDialogTask interface

The assessExecutability method is a task-specific method for informing the dialog manager how likely it is a task will succeed. For example, sending an email may be guarenteed to fail if the device does not have working internet. For situated agents such as robots, there is always some non-negligible chance that an attempted physical action will fail. The dialog manager will incorporate these probabilities into its decision-making process. Intuitively, it will hesitate to perform actions which are likely to fail, and eagerly execute actions which are likely to succeed.

The execute method executes the actual task. This will call developer-defined functions and libraries to actually send emails, reserve meeting rooms, etc.

The status method checks a task's status, which can be one of the following:

- SUCCESSFULLY_COMPLETED
- CURRENTLY_EXECUTING_BLOCKING
- CURRENTLY_EXECUTING_NOT_BLOCKING
- FAILED

When a task is blocking, the dialog system is temporarily de-activated. In this tutorial, we assume that a separate module is called for email dication, so the send-email task is blocking.

Define interfaces to External Databases

Classes in the ontology correspond to supporting databases.

There are two types of supporting databases, corresponding to two primary functions:

- **Long-term Memory Database:**Store objects from interactions in long-term memory to keep reasoning in working memory tractable while allowing for possible retrieval at a later date
- **Reference Database:** Store and index a large collection of reference objects (the standard use paradigm of a database)

By default, every class defined in the ontology has no supporting reference database, and an independent long-term memory database. If the developer wants to provide a reference database they may do so. The developer can decide weather or not the dialog system is allowed to insert to that database. If the dialog system is not allowed to insert, then a separate long-term memory database will be automatically created. Otherwise, the reference database will provide both long-term memory and reference functions.

Reference databases must be Sesame RDF databases. Many existing databases are in relational format and are accessed via SQL. These databases can be represented as RDF databases, as is described in A Survey of Current Approaches to Mapping of Relational Databases to RDF. YODA requires a map between slots and properties which are shared by the domain ontology and the database schema. YODA only allows a one-to-one mapping of properties and slots, but allows certain properties to exist only in the reference database.

Any database interface that does not have the default setting is defined by modifying the yoda/interfaces/DatabaseInterfaces.java file.

Define interfaces to Non-dialog actions

Similar to the database interface definitions, the developer can specify action interfaces to specify what information is needed to actually perform an action.

Build a lexicon for NLG

Initial linguistic information should be associated with each new class and instance. The two figures below show examples of the linguistic information associated with the objects specified above. The complete example is given in the attached owl file.

The program TestLexicalEntry.java generates sample sentences based on the lexicon provided by the developer. The developer can use this program to test that their lexicon generates sensible sentences and covers the full variety of things the developer expects to be said about the object. Call it from the command line using: java TestLexicalEntry.java -e [entryID]

Train SLU and speech modules with artificial data

To build an initial SLU component, YODA takes the generated lexicon and creates a large training set. It adds artificial noise to the training set for robustness, and extends the training set using wordnet synonyms. It then trains a state-of-the-art SLU component based on this artificial data.

To build an initial SLU component, run java GenerateInitialSLUComponent.java . Include -v in the command line arguments after the filename to see cross-validation results within the auto-generated corpus.

The training data set is retained in yoda/SLU/artificial_train.txt, and the domain-specific annotation scheme is described in yoda/SLU/annotation_scheme.txt . As real data is collected, the SLU component can be improved by incorporating annotated real-world examples.

To build an improved SLU component, put labeled real-world data in yoda/SLU/real_data.txt in the same format as artificial_rain.txt, then run: java GenerateImprovedSLUComponent.java . Include a -v in the arguments after the filename to see validation results and a comparison between the initial and the improved systems.

It will likely be the case that as real data is collected, the developer will discover that the domain and lexicon definitions should be extended to improve coverage of users actully say. We recommend that this be done by extending the ontology rather than simply training the improved model. The purpose of the improved model is to improve the component's likelihood model for different utterances, not to compensate for poor coverage.

To generate a language model based on the automatically generated sentences and the real data examples, run the script build_language_model.sh .

Build sensor interfaces

YODA dialog systems support situated interaction by defining sensor interfaces. Sensors can report the existance of new entities, and their properties and relations. The interfaces define a message protocol used to receive updates from the sensors.

Some of the information provided in the protocol includes:

- Expected framerate
- Expected consistency of the sensor readings over time
- Type of confidence markup (N-best, Bayes net, full CPT)
- Overlap interface (Define the overlap between this sensor and other sensors)
- Relevance interface (Choose a strategy for defining how relevance to the ongoing discourse should be tracked for objects in this sensor stream)
- Expected recall of objects (effects dialog strategy)
- Expected precision (of some slot?)

Sensor belief tracking in Yoda

YODA designers will explore belief-tracking and sensor fusion approaches that generalize to dynamic situated domains.

Long-term Memory and Discourse Relevance in YODA

YODA designers will explore different strategies for tracking objects' relevance to an ongoing discourse, and for using that relevance to move information between long-term and working memory. The different properties of different sensor types will be better suited for different relevance strategies.

```java
public class SendEmailTask implements NonDialogTask {
    private static Integer instanceCounter = 0;
    private static Map<String, TaskStatus> executionStatus =
        new HashMap<>();
    private static NonDialogTaskPreferences preferences =
        new NonDialogTaskPreferences(true, 1, 20, 20,
            new HashSet<>(Arrays.asList("hasToPerson")));

    @Override
    public NonDialogTaskPreferences getPreferences() {
        return preferences;
    }

    @Override
    public double assessExecutability(SemanticsModel taskSpec) {
        // this works because assessExecutability is never
        // called unless all the required slots are present
        return 1.0;
    }

    @Override
    public String execute(SemanticsModel taskSpec) {
        System.out.println("Executing task: send email");
        System.out.println(taskSpec);
        String ans = "SendEmailTask:"+instanceCounter.
            toString();
        instanceCounter += 1;
        // There is no real implementation, so we just set
        // the status to successfully completed
        executionStatus.put(ans,
            TaskStatus.SUCCESSFULLY_COMPLETED);

        /*
         * In the envisioned implementation, a separate
         * dictation program would run while the other
         * program is run, the dialog manager should block, as
         * in the following commented lines
         */
        // executionStatus.put(ans,
        //    TaskStatus.CURRENTLY_EXECUTING_BLOCKING);
        // dictationProgram.run();

        System.out.println("taskID: "+ans);
        return ans;
    }

    @Override
    public TaskStatus status(String taskID) {
        return executionStatus.get(taskID);
    }
}
```

Figure 4: The SendEmailTask implementation

```java
public class CreateMeetingTask implements NonDialogTask {
    private static Integer instanceCounter = 0;
    private static Map<String, TaskStatus> executionStatus =
        new HashMap<>();
    private static NonDialogTaskPreferences preferences =
            new NonDialogTaskPreferences(false, 1, 20, 15,
                    new HashSet<>(Arrays.asList(
                        "hasMeeting.hasTime",
                        "hasMeeting.hasPlace",
                        "hasMeeting.hasPerson")));

    @Override
    public NonDialogTaskPreferences getPreferences() {
        return preferences;
    }

    @Override
    public double assessExecutability(SemanticsModel taskSpec) {
        return 1.0;
    }

    @Override
    public String execute(SemanticsModel taskSpec) {
        System.out.println("Executing task: create meeting");
        System.out.println(taskSpec);
        String ans = "CreateMeetingTask:"+instanceCounter.
            toString();
        instanceCounter += 1;
        // There is no real implementation, so we just set the
        // status to successfully completed
        executionStatus.put(ans,
            TaskStatus.SUCCESSFULLY_COMPLETED);
        System.out.println("taskID: "+ans);
        return ans;
    }

    @Override
    public TaskStatus status(String taskID) {
        return executionStatus.get(taskID);
    }
}
```

Figure 5: The SendEmailTask implementation