trindikit.py: An open-source Python library for developing ISU-based dialogue systems

Peter Ljunglöf

Department of Philosophy, Linguistics and Theory of Science
University of Gothenburg, Sweden
peb@ling.gu.se

Abstract. TrindiKit is one of the main tools for developing ISU-based dialogue systems, but it is implemented in a non-free dialect of the programming language Prolog. Therefore we have translated the TrindiKit toolkit into an open-source Python package. We have tried to remain close to the original TrindiKit formulation, while making the most of Python classes and objects.

1 Background

1.1 The ISU approach to dialogue management

The information state update (ISU) approach is a framework for specifying, implementing and reasoning about dialogue theories. The only thing that is common between different ISU theories is the idea of an *information state* which is modified by *update rules* triggered by *dialogue moves*. In essence, an ISU dialogue theory consists of:

- A characterisation of the information states, which describes all aspects of common context as well as internal motivating factors.
- A set of dialogue moves. These are generally associated with utterances, or parts of utterances, by the dialogue participants.
- A set of update rules, that govern the updating of the information state. An update rule consists of a list of preconditions on the information state, and a list of effects that modify the information state.
- An update strategy for deciding which rule or rules to select at a given point.

Note that this article is not an introduction to the ISU approach, instead we refer to [1,2,3] for more information. However, one of the main advantages with ISU is that it makes it possible to separate the dialogue manager from the domain knowledge, which makes it easier to adapt an existing dialogue system to new domains and languages.

1.2 TrindiKit

TrindiKit [4] is one of the main tools for developing ISU-based dialogue systems. A similar toolkit, but with a less expressive update language, is Dipper [5].

Examples of ISU theories which have been implemented in TrindiKit include the GoDiS/IBiS dialogue managers [2], and the Poesio-Traum theory (PTT) [6].

TrindiKit is a module-based architecture, where the modules consist of different components in a dialogue system, such as speech recognition, speech synthesis, and, of course, the ISU dialogue manager. The modules can be activated in sequence according to a control algorithm, or they can all be active all the time asynchronously. They communicate with each other by reading and writing on a part of the dialogue manager's information state.

TrindiKit is implemented in Sicstus Prolog, and uses OAA, the Open Agent Architecture [7], for asynchronous agent communication. The implementation works fine, and has been used by several research groups throughout the world. There are however some issues with the implementation, which can become problematic for dialogue system developers:

- Prolog is not a common programming language outside the research community, and OAA is not a common framework for asynchronous agent communication. Most software developers have never used Prolog or OAA, and since logic-based languages are declarative, they are probably not similar to any language that the developer knows about.
- Also, while being free and open-source itself, OAA requires Sicstus Prolog, which is a non-free Prolog implementation. Furthermore, OAA is no longer in active development, which means that there will presumably be problems on future computers and operating systems.

To make TrindiKit and the ISU approach more accessible to people outside academia and on standard hardware, we are now redeveloping TrindiKit in the more commonly known programming language Python. Apart from being an object-oriented scripting language, it is also used to implement NLTK, the Natural Language Toolkit, a large, open-source library for natural language processing. NLTK contains a large number of modules for natural language processing such as tagging, parsing and semantic interpretation, as well as many freely available corpora. Additionally, there is an accompanying book for learning NLP using the NLTK library [8]. The `trindikit.py` module is designed to be used together with NLTK, but it can also be used on its own.

2 trindikit.py: A toolkit for creating dialogue managers

Creating a dialogue system using TrindiKit is a two-step process. In Python this is naturally implemented using subclassing and creating instances. First we have to implement a dialogue manager, which is independent of the domain-specific details of a dialogue system. This is implemented as a subclass of the `trindikit.py` class DialogueManager. From this class we can then create an instance by giving domain-specific information, such as an utterance grammar and a query database.

The idea behind dividing the dialogue system into two parts (the class and the instance), is that the dialogue theory is designed, implemented and maintained by a dialogue specialist, while the particular dialogue systems will be

implemented by other people who are not specialists in dialogue management issues. Therefore the implemented dialogue manager should be a "black box", with an intuitive API for developing the final dialogue system.

In this section we describe the first step, the specification of the dialogue manager. In TrindiKit, a dialogue manager consists of (1) a number of dialogue modules, (2) the information state specification, (3) a dialogue move taxonomy, (4) update rules, and (5) a control algorithm.

Note that this is not a description of how to implement a dialogue system for a particular domain, but instead of how to implement a ISU dialogue theory. As an example we show how to implement a very simple variant of the IBiS dialogue manager [2]. Later, in section 3, we describe what remains to be done to get a working dialogue system.

In the following, we presuppose basic knowledge of Python and Python concepts such as classes, instances and decorators.

2.1 The information state

The information state is stored as instance variables in the final dialogue manager object. The exact names and data structures of these information state variables depend on the dialogue theory, but this is a common division:

- The information state for the dialogue manager is a record called IS. As an example, we use an information state taken from [1,3], containing shared and private beliefs (sets of propositions), an agenda (a stack of things to do), the qud (a stack of questions currently under discussion), and lm (the set of dialogue moves in the latest utterance).
- The dialogue modules communicate with each other by reading and writing to the *module interface variables*. In our example we use LATEST_MOVES and NEXT_MOVES which are sets of dialogue moves, representing what the user just said and what the system is about to say.
- External *resources* are also stored as instance variables. In our example, the domain model, the parser and the system utterances are located in the instance variables DOMAIN, PARSER and GENERATOR.

This information state is described as follows (where the non-standard Python classes record and stack are defined in `trindikit.py`):

```
class Infostate:
    def __init__(self, domain, parser, generator):
        self.IS = record(private = record(beliefs = set(), agenda = stack()),
                         shared = record(beliefs = set(), qud = stack(), lm = set()))
        self.LATEST_MOVES = set()
        self.NEXT_MOVES = set()
        self.DOMAIN = domain
        self.PARSER = parser
        self.GENERATOR = generator
```

2.2 Dialogue modules

Apart from having a module for updating the information state according to our preferred dialogue theory, we need additional non-dialogue modules. We need at least one module for speech interpretation (translating user speech into dialogue moves) and one for speech generation (translating dialogue moves into system utterances), but these are often split into several different modules. The modules never interact directly with each other, but always by reading and writing the module interface variables.

The following example generation module reads the next moves to be performed, translates them into a string and calls the speech synthesiser (which in this case is the open-source package `speech.py`):

```
class Generate:
    def generate_speech(self):
        output = self.GENERATOR.generate(self.NEXT_MOVES)
        speech.say(output)
```

2.3 Dialogue moves

Dialogue moves are actions associated with utterances. A single utterance may be associated with several moves, simultaneous or ordered in time. The specific dialogue theory specifies what kinds of dialogue moves there are, and what their content is. All dialogue moves should be subclasses of the base class Move. In our example theory, we have the moves Ask(q) and Answer(a), where q is a question and a is an answer. In addition to these, there are the plan items Findout(q) and ConsultDB(q) which are only used within dialogue plans.

2.4 Update rules

An update rule in TrindiKit consists of a precondition and an effect. The precondition tests constraints on the information state, and it can bind variables which can be used in the effect. If the precondition matches (with some possible bindings), the effect is executed and is allowed to modify the current state. As an example, the following update rule from [3] integrates a user answer, if it is relevant to some question under discussion, by adding the resolved proposition to the shared beliefs:

Integrate User Answer

PRECONDITION:	EFFECT:
$\text{answer}(a) \in$ SHARED.LM	$p =$ DOMAIN :: $\text{reduce}(q, a)$
$q = \text{fst}(\text{SHARED.QUD})$	$\text{add}(\text{SHARED.BELIEFS}, p)$
DOMAIN :: $\text{relevant}(a, q)$	

In `trindikit.py`, an update rule is implemented as an ordinary function taking information state variables as arguments. To turn it into an update rule acting on a DialogueManager instance, we apply the @update_rule decorator to the function

definition. First, the precondition should be tested by applying the @precondition decorator to its definition. A precondition is implemented as a generator function yielding all possible bindings, which is similar to the backtracking search strategy used by Prolog. The effect then takes the first yielded binding as argument and modifies the information state. If the precondition does not yield anything, a failure exception is raised.

To implement the example update rule, we notice that a and q both occur in the precondition and in the effect, whereas p only occurs in the effect. This means that the precondition needs to yield a and q, and the effect takes a and q as arguments:

```
@update_rule
def integrate_user_answer(IS, DOMAIN):
    @precondition
    def binding():
        for move in IS.shared.lm:
            if isinstance(move, Answer):
                question = IS.shared.qud.first()
                if DOMAIN.relevant(move.content, question):
                    yield record(ans=move.content, que=question)
    proposition = DOMAIN.reduce(binding.que, binding.ans)
    IS.shared.beliefs.add(proposition)
```

The @update_rule decorator turns the definition into a function taking one single argument, an Infostate having the instance variables IS and DOMAIN. The @precondition decorator executes the precondition, extracts the first yielded result, and assigns the result of the execution to the variable binding. If the precondition fails, an exception is raised, which can be captured by the control algorithm.

Several update rules can be grouped together by creating an *rule group*, corresponding to in TrindiKit's rule classes, which executes the first rule whose precondition succeeds. The following are the two rule groups that are defined in the example dialogue manager:

```
integrate = rule_group(integrate_answer, integrate_ask)
select = rule_group(select_answer, select_ask)
```

2.5 The control algorithm

The control algorithm is responsible for calling different dialogue modules and update rules. An update rule can either be obligatory, optional, or repeated until it fails. The corresponding methods in the DialogueManager class are do(), maybe(), and repeat(), respectively.

Assuming that we have dialogue modules for speech interpretation and speech generation, we can define a very simple control algorithm, which integrates the user's dialogue moves into the information state, tries to query the domain model if possible, and then selects a single system utterance. The system utters this and then listens for the user's next utterances, after which it starts over again:

```
class SimpleManager(Interpret, Generate, Infostate, DialogueManager):
    def control(self):
        while True:
            self.repeat(integrate)
            self.maybe(query_domain)
            self.do(select)
            self.generate_speech()
            self.interpret_speech()
```

This, together with the definitions of Interpret, Generate and Infostate, is now an implementation of our example dialogue theory. To create and run a functioning dialogue system is just a matter of creating an instance of the SimpleManager class. Of course, things are never that simple. In our example case, we have to create appropriate classes for the domain, the parser and the generator.

2.6 The domain, database, parser and generator classes

The update rules in the SimpleManager don't say anything about the semantics of dialogue moves, except for that a Domain instance has to provide the methods relevant(), reduce(), find_plan() and consultDB(). The first two methods check if answers are relevant to questions, and combine question and answers into propositions. The final two methods find appropriate dialogue plans, and consult the domain database.

To simplify things for the dialogue system implementor, we define the Domain method __init__() to take a description of the predicates in the domain as argument. From this it constructs the domain semantics and the methods for checking relevance and doing reduction. We also provide the methods add_plan() and add_dbentry(), which add dialogue plans and database entries to the domain.

In a similar way do the Parser and the Generate classes only require that instances have the methods parse() and generate(), respectively. The creation of these instances can also be simplified by providing tailor-made methods.

3 Implementing the final dialogue system

With the implementation of SimpleManager as described in section 2, we can implement a working dialogue system by creating instances of the classes Domain, Parser and Generator. Our example system is a very simple travel agent who can only answer questions about the price of trips.

The domain First we initialise the domain by specifying its predicates, and the possible arguments they take:

```
cities = 'london', . . ., 'paris'
days = 'monday', . . ., 'friday'
travel_domain = Domain({'to':cities, 'from':cities, 'when':days, 'howmuch':int})
```

Then we add the dialogue plans, which in this case is only one:

```
travel_domain.add_plan('?x.price(x)', [Findout('?x.to(x)'), Findout('?x.from(x)'),
                                       Findout('?x.when(x)'), ConsultDB('?x.price(x)')])
```

This plan says that, to be able to answer a price question, the system first needs to find out how and when the user wants to travel, after which it can consult its database to get the price. Finally, we enter the price information for different trips as database entries:

```
travel_domain.add_dbentry({'price':232, 'from':'berlin', 'to':'paris', 'day':'monday'})
travel_domain.add_dbentry({'price':340, 'from':'berlin', 'to':'london', 'day':'friday'})
```

The parser and generator One possibility out of many, is to derive the parser from a NLTK feature structure grammar [8, chapter 9], and to specify the system utterances as a mapping from dialogue moves to strings:

```
travel_parser = Parser('file:travel_english.fcfg')
travel_generator = Generator({Ask('?x.from(x)'): 'From where do you want to leave?',
                              Ask('?x.to(x)'): 'To where do you want to go?', . . .})
```

The final dialogue system Finally, after specifying the domain, parser and generator, all we have to to is to create a dialogue manager instance and run it:

```
travel = SimpleManager(travel_domain, travel_parser, travel_generator)
travel.run()
```

4 Discussion

We have translated the TrindiKit toolkit for developing ISU dialogue systems, into an open-source Python package. We have tried to remain close to the original TrindiKit formulation, while making use of the object-oriented Python paradigm.

System development To develop a dialogue system, the dialogue manager is first specified as a subclass of the abstract base class DialogueManager. This subclass is supposed to be a domain-independent implementation of some dialogue theory, such as GoDiS/IBiS [2] or PTT [6]. Then, several different dialogue systems can be created as instances of the dialogue manager, by giving the relevant domain-dependent information as arguments.

Of course, a real-sized dialogue system will be larger than the toy example given in section 3. Not only will the dictionaries be larger, but the domain and the grammar will also be more complicated. Probably the domain specification will be put in a Python module of its own, perhaps using semantic models from NLTK, and the database queries will be moved into a SQL database, or an OWL ontology. The interpretation grammar might be split into several modules, making use of NLTK's taggers, chunkers and semantic grammars. It can be specified in a different grammar formalism, such as a probabilistic grammar, a dependency grammar, or a grammar in the Grammatical Framework [9,10]. The generation component will probably be more advanced than just a dictionary lookup. It could for example make use of the domain ontology and the interpretation grammar to generate system utterances.

Object-oriented vs. logic-based One thing we have not investigated yet is the fact that we have changed the underlying paradigm from logic-based Prolog, to object-oriented Python. It would be interesting to examine how object-oriented dialogue models [11,12] can be adapted to this framework.

Conclusion The main thing to note with the specification is that the internals of the dialogue manager are completely hidden from the dialogue system. This means that the developer does not have to be an expert on dialogue. But it also means that the dialogue behaviour can be modified without changing the domain specification, as long as the API remains the same. This separation of dialogue and domain is arguably the main advantage of the ISU approach.

References

1. Larsson, S., Traum, D.: Information state and dialogue management in the TRINDI dialogue move engine toolkit. Natural Language Engineering **6** (2000) 323–340
2. Larsson, S.: Issue-based Dialogue Management. PhD thesis, Department of Linguistics, University of Gothenburg, Sweden (2002)
3. Traum, D., Larsson, S.: The information state approach to dialogue management. In Smith, Kuppevelt, eds.: Current and New Directions in Discourse and Dialogue. Kluwer Academic Publishers (2003) 325–353
4. Larsson, S., Berman, A., Bos, J., Grönqvist, L., Ljunglöf, P., Traum, D.: TrindiKit 2.0 manual. Deliverable D5.3, TRINDI Project (2000)
5. Bos, J., Klein, E., Lemon, O., Tetsushi, O.: Dipper: Description and formalisation of an information-state update dialogue system architecture. In: Proc. 4th SIGdial Workshop on Discourse and Dialogue, Sapporo, Japan (2003)
6. Matheson, C., Poesio, M., Traum, D.: Modelling grounding and discourse obligations using update rules. In: Proc. NAACL'00, 1st Meeting of the North American Chapter of the Association for Computational Linguistics, Seattle, WA (2000)
7. Martin, D.L., Cheyer, A.J., Moran, D.B.: The Open Agent Architecture: A framework for building distributed software systems. Applied Artificial Intelligence **13** (1999) 91–128
8. Bird, S., Klein, E., Loper, E.: Natural Language Processing with Python. O'Reilly (2009) Also available at http://www.nltk.org/book.
9. Bringert, B., Cooper, R., Ljunglöf, P., Ranta, A.: Multimodal dialogue system grammars. In: Proc. Dialor'05, 9th Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France (2005)
10. Ljunglöf, P., Larsson, S.: A grammar formalism for specifying ISU-based dialogue systems. In: Proc. GoTAL'08, 6th International Conference on Natural Language Processing. Number 5221 in Springer LNCS/LNAI, Gothenburg, Sweden (2008)
11. Abella, A., Gorin, A.L.: Construct algebra: Analytical dialogue management. In: Proc. ACL'99, 37th Annual Meeting of the Association for Computational Linguistics, College Park, MD (1999)
12. O'Neill, I.M., McTear, M.F.: Object-oriented modelling of spoken language dialogue systems. Natural Language Engineering **6** (2001) 341–362