Representing Act-Topic-based Dialogue Phenomena

Hans Dybkjær[1] and Laila Dybkjær[2]

[1]Prolog Development Center A/S (PDC),
H. J. Holst Vej 3C-5C, 2605 Brøndby, Denmark, dybkjaer@pdc.dk
[2]Nislab, University of Southern Denmark (SDU),
Campusvej 55, 5230 Odense M, Denmark, laila@nis.sdu.dk

Abstract. We examine phenomena in spoken human computer dialogues and suggest possible formalisations. The work is a step towards automating spoken dialogue systems assessment.

Keywords. Spoken dialogue, act-topic, structure analysis, transactions.

1 Introduction

Dialogue smoothness and transaction success rate are important SDS usability evaluation criteria but are costly to measure manually. Automating the measurement process would be of great benefit to the SDS community.

We suggest a two-step approach to the automatic annotation of act-topics and, eventually, of transactions. *First*, basic, context-independent act-topic annotation is added to all system and user utterances (Figure 1, left). Basic acts include `inform`, `accept`, `reject`. *Second*, basic acts are combined into composite acts and then further combined into transaction segments tagged with success or failure (Figure 1, right).

This paper investigates the second step: what is needed to automate act-topic based transaction structure annotation of dialogues between users and spoken dialogue systems (SDSs), such as a frequently asked questions (FAQ) system [Dybkjær and Dybkjær 2004], a flight ticket reservation system [Bernsen et al. 1998], and a train timetable information system [Aust et al. 1995]. As the formal vehicle for deriving composite acts we use rewrite rules with unification and supplementary constraints.

```
.s: .inform {N.employee, N.leave
…}
    "You can … choose between:
     'employee' 'on leave' …"
.u: .inform {N.student}
    "I'm a student"
.s: .inform {N.menu}
    "Did you ask for – Main menu?"
.u: .inform {N.student}
    "Student"
.s: .inform {V.student, V.su …}
    "If you are a student and
     receive SU, you may …"
```

```
.s: .success {N.student}
  <- success2
  .u: .request {N.student}
    <- sequenceRequest
    .u: request {N.student}
      <- request2
      .s: inform {N.employee, N.leave …}
      .u: inform {N.student}
    .u: request {N.student}
      <- request2
      .s: inform {N.menu}
      .u: inform {N.student}
  .s: .inform {V.student, V.su …}
```

Figure 1: Example dialogue, annotated with acts and topics.

2 Dialogues, turns, and moves

A *dialogue* is a sequence of *moves* where each move corresponds to one act and a set of topics for one speaker (Figure 2). An *utterance* is a sequence of moves of one speaker. A *turn* is an utterance that further satisfies that if other moves occur at the ends, then these moves belong to other speakers. In Figure 2, e.g., the example has one user utterance that is also a turn, and there are two system moves that may make one or two utterances and precisely one turn.

```
dialogue = move*
move = who : act topics ["text"]
who = .u | .s
act = .identifier
topics = { topic* }
topic = distinction.identifier
distinction = T | N | V
```

Example:

```
.s: .pause {}
.s: .inform {T.more}
    "Do you want more?"
.u: .accept {}
    "yes"
```

Figure 2: Formal structure of a dialogue.

3 Dialogue structure via rewrite rules

Rewrite rules define an acyclic graph which may be seen as the dialogue structure. Each rule takes a move pattern and produces a new sequence of moves. A *pattern* is a list of utterances but may contain variables for *who*, *act*, *topics*, and *topic*, cf. Figure 3.

```
rule = rule identifier
         move* <- move*
         [ where condition* ]
       end rule
move M = who : act topics
who W = varVal
act A = varVal
topics Ts = { topic* } | varVal
topic T = varVal
varVal = [Type]var | [Type]val
var = _identifier
val = .identifier
condition = varVal operator varVal
operator = =|!= |in|not-in | <
```

Example:

```
rule select1
  _y: .select Ts_a
    <-
    _x: .inform Ts_a
    _y: .accept {}
  where
    _x != _y
end rule
```

Result when applied to example (Figure 2):

```
.s: .pause {}
.u: .select {T.more}
  <- select1
  .s: .inform {T.more}
      "Do you want more?"
  .u: .accept {}
      "yes"
```

Figure 3: Formal structure of rules and their application.

4 Analysing dialogue phenomena

The minimum expressiveness of the above rules would allow for one speaker, one act, one topic, and no constraints. The table below explains and exemplifies different needed extensions to the minimum expressiveness based on various dialogue phenomena. Rule references are to Appendix A.

Different topics

Allowing several topics enables detection of two moves including the same topic.

See `rule segment`

Different acts

```
x: Are you going to Copenhagen?
y: Yes
```

Allowing *different acts* instead of only one, enables distinction among certain patterns, such as a select pattern which may consist of an inform act telling about an option followed by an accept act of this option.

See `rule select0`

Differentiating speakers

```
s: Are you going to Copenhagen?
u: Yes
```

To distinguish that moves are by different speakers, an *inequality operator* is needed.

See `rule select1`

More topics in a move

```
x: Do you want to know about when the money is paid, transfer of money,
   or payment in general?
y: Payment in general.
```

A select rule matching this example must express that speaker `y` states one of the topics offered by speaker `x`. A member operator "*`T in Ts`*" is added as a constraint.

See `rule select2`

More topics in different moves

```
.s: .inform {T.paymentWhen, T.moneyTransfer, T.paymentGeneral}
    "Do you want to know about when the money is paid, transfer of money,
     or payment in general?"
.u: .inform {T.paymentGeneral, T.friend}
    "Payment in general, my friend."
```

Here we need to express that the same topic occurs in two different lists. We do so by allowing variables to be introduced in the constraints, too, and not only in the pattern.

See `rule select2a`

Rejecting topics

```
x: Do you want to know about when the money is paid, transfer of money,
   or payment in general?
y: My employer is bankrupt.
```

Symmetrically to selecting a topic a speaker may reject the offered topics by requesting a new topic, so we add the "*`T notT-in Ts`*"operator.

See `rule request`

Distinguishing names and values

```
.u: .inform {N.phone}
    "Your phone number?"
.s: .inform {N.phone}
    "Phone number"
```

```
.u: .inform {N.phone}
    "Your phone number?"
.s: .inform {V.phone}
    "Phone 48204910"
```

In order to distinguish the analyses of these two examples, we must distinguish topic names (N) and values (V). By a topic *name* we understand the mentioning of a topic, e.g. in terms of a user requesting information about a certain topic. By a topic *value* we understand details about a certain topic, e.g. the system informing about a topic name selected by the user.

See `rules select2, answer, success1 and success2`

Patterns across turns not using all the turn moves

```
0 .u: .inform {V.aalborg, V.tomorrow}
      "I want to go to Aalborg tomorrow."
1 .s: .inform {V.aalborg, V.aarhus}
      "Did you say to Aalborg or to Aarhus?"
2 .u: .inform {V.aalborg}
      "To Aalborg."
3 .s: .inform {V.tomorrow}
4 .s: .inform {V.aalborg}
      "Are you leaving tomorrow for Aalborg?"
5 .u: .accept {}
      "Yes."
```

The dialogue fragment above contains a success in selecting travel destination and date. The moves 1+2 may be reduced to `select {V.aalborg}` which potentially enables us to apply the `success1` rule. However, the subsequent utterance is annotated with the moves in a wrong order for this. There is no inherent reason why the order of moves within an utterance or turn should be important (at the level of analysis we do), and the equivalent formulation of move 4 above "`Aalborg. Are you leaving tomorrow?`" would naturally have made the annotator list the moves in the reverse order.

So we will allow patterns to match moves within turns in any order, leaving unused moves if the turns are at the ends of the pattern, otherwise dropping the unused moves.

Ontological relations

```
.s: .inform {N.travel, N.from}
    "Where does the travel start?"
.u: .inform {V.place}
    "Copenhagen"
```

When annotating each move independently of the context it becomes ambiguous what a topic value refers to. E.g. the place “Copenhagen” may be departure or destination city. To be able to automatically relate the question and the reply in such situations we need to introduce the *sub-topic relation* `T1 < T2`.

See rule `selectSub1`

Meta-communication and multi-level rule applications

In Figure 1 it seems fair to count a success and no failures, and to count one meta-exchange which is negative for the smoothness. If the two requests had been divided by several exchanges or even a full transaction on another topic, it is less obvious that the two requests should be counted as one, leading to one success. Part of a solution to handling meta-communication involves the division of rules into several sets that are applied successively. This reflects that naturally one would first do simple rewrites like detecting request and select, then handle meta-communication, then transactions, and finally issues like summarising feedback.

See rule `sequenceRequest`

Summarising feedback

```
.s: .inform {V.from, V.to, V.hour}
    "Es gibt die folgende Verbindung: Mit der S-Bahn Abfahrt in Berlin
    Hauptbahnhof um fuenfzehn Uhr vierundzwanzig Ankunft in Berlin-Zoo um
    fuenfzehn Uhr einundvierzig dort weiter mit … dort weiter mit Inter-
    city sechs fuenf zwei Abfahrt um zwanzig Uhr einundfuenfzig Ankunft in
    Darmstadt Hauptbahnhof um einundzwanzig Uhr neun"
```

We call such information *summarising feedback*. It is fairly common in information and booking systems. Often systems implementing this have "one call – one task" dialogues, and a simple measure of transaction success is to call it a success if the system reaches this state, and otherwise a failure. However, at least on two points this is problematic:

- The user may disagree in the summarisation, claiming something to be wrong.
- It provides no information on dialogue smoothness up until this point.

By instead using rules like those in Appendix A and assuming the `reject3` rule is applied before the `successSummary` rule, we may get a success/failure annotation that deals with the above two bullet points (`reject3` will block `successSummary`).
See rule `reject3` and `successSummary`

5 Conclusion

By applying subsequent levels of act-topic rewrite rules we can analyse a set of dialogue phenomena occurring in typical SDSs, eventually leading to automatic detection of non-smoothness and of transaction successes and failures.

Compared to other work, the two key distinguishing features of our analysis are *automation* and *act-topic structures*. Many other papers discuss how to find acts and/or topics [Heeman et al. 1998, Jurafsky et al. 1997], often based on statistical methods, but are not concerned with the further structural analysis of the dialogue structure. The probably most dominant discourse structure theory, RST (Rhetorical Structure Theory, [Mann and Thomson 1987]), is not aimed at computational analysis.

Much work remains to be done. There are unanalysed issues regarding in particular smoothness and summarising feedback, common to which is that they concern phenomena distributed over large parts of the dialogue instead of being locally (and continuously) defined. Other issues also needing further analysis include task dependence of rules, summaries not including all information, information stated in disguise, and inexact matches. The rules must be tested on larger sets of dialogues of different type. An automatic act-topic annotation parser must be made in order to achieve full automation.

Note that our approach only considers structure. For instance, the correctness of summarising feedback is not considered.

References

[Aust et al. 1995] Harald Aust, Martin Oerder, Frank Seide, and Volker Stenbiss: The Philips Automatic Train Timetable Information System. *Speech Communication*, 17, 249-262, 1995.

[Bernsen et al. 1998] Niels Ole Bernsen, Hans Dybkjær and Laila Dybkjær: *Designing Interactive Speech Systems. From First Ideas to User Testing.* Springer Verlag 1998.

[Dybkjær and Dybkjær 2004] Hans Dybkjær and Laila Dybkjær: Modeling Complex Spoken Dialog. *Computer*, IEEE, August 2004, 32-40.

[Heeman et al. 1998] Peter A. Heeman, Donna Byron, and James F. Allen: Identifying Discourse Markers in Spoken Dialog. *AAAI Spring Symposium on Applying Machine Learning and Discourse Processing*, Stanford, March 1998.

[Jurafsky et al. 1997] Daniel Jurafsky, Rebecca Bates, Noah Coccaro, Rachel Martin, Marie Meteer, Klaus Ries, Elizabeth Shriberg, Andreas Stolcke, Paul Taylor, and Carol van Ess-Dykema: *Automatic Detection of Discourse Structure for Speech Recognition and Understanding*. Proc. of IEEE Workshop on Speech Recognition and Understanding, 1997, 88-95.

[Mann and Thompson 1987] William C. Mann and Sandra A. Thompson: Rhetorical Structure Theory: Description and Construction of Text Structures. In Gerard Kempen (ed.): *Natural Language Generation. New results in artificial intelligence, psychology and linguistics.* NATO ASI series E 135, Chapter 7. The Netherlands, Martinus Nijhoff Publishers, 1987.

Appendix A Example rules

```
rule segment
  _y: .any {T_a}
    <-
    _x: .any {T_a}
    _y: .any {T_a}
end rule
```

```
rule select0
  _y: .select {T_a}
    <-
    _x: .inform {T_a}
    _y: .accept {}
end rule
```

```
rule select1
  _y: .select {T_a}
    <-
    _x: .inform {T_a}
    _y: .accept {}
  where
    _x != y
end rule
```

```
rule select2
  _y: .select {T_b}
    <-
    _x: .inform Ts_a
    _y: .inform {T_b}
  where
    _x != y
    _b in _a
end rule
```

```
rule select2a
  _y: .select {T_c}
    <-
    _x: .inform Ts_a
    _y: .inform Ts_b
  where
    _c in _a
    _c in _b
    _x != y
end rule
```

```
rule request
  _y: .request {T_b}
    <-
    _x: .inform Ts_a
    _y: .inform {T_b}
  where
    _x != y
    _b not-in _a
end rule
```

```
rule answer
  _y: .request {N_b}
  _x: .inform Vs_a
    <-
    _y: .inform {N_b}
    _x: .inform {V_b}
  where
    _x != _y
end rule
```

```
rule success1
  _y: .success {N_b}
    <-
    _x: .select {N_b}
    _y: .inform Vs_a
  where
    _b in Vs_a
    _x != _y
end rule
```

```
rule success2
  _y: .success {N_b}
    <-
    _x: .request {N_b}
    _y: .inform Vs_a
  where
    _b in Vs_a
    _x != _y
end rule
```

```
rule request2
  _y: .request {V_b}
    <-
    _x: .inform Ts_a
    _y: .inform {V_b}
  where
    _x != y
    _b not-in _a
end rule
```

```
from-place < place
rule selectSub1
  _y: .select {N_a}
  <-
    _x: .inform {N_a}
    _y: .inform {T_b}
  where
    _a < _b
    _x != _y
end rule
```

```
rule sequenceRequest
  _y: .request {T_a}
    <-
    _y: .request {T_b}
    _y: .request {T_b}
end rule
```

```
Rule reject3
  _u: .reject {V.toPlace,
               V.fromPlace,
               V.departureTime}
    <-
    _s: .inform {V.toPlace,
                 V.fromPlace,
                 V.departureTime}
    _u: .reject
end rule
```

```
rule successSummary
  _y: .success {N.travel}
    <-
    _u: .success {V.departureTime}
    _u: .success {V.fromPlace}
    _u: .success {V.toPlace}
    _s: .inform {V.toPlace,
                 V.fromPlace,
                 V.departureTime}
end rule
```