Dialogue management for automatic troubleshooting and other problem-solving applications

Johan Boye
TeliaSonera R&D
Vitsandsgatan 9
12386 Farsta, Sweden
johan.boye@teliasonera.com

Abstract

This paper describes a dialogue management method suitable for automatic troubleshooting and other problem-solving applications. The method has a theorem-proving flavor, in that it recursively decomposes tasks into sequences of sub-tasks and atomic actions. An explicit objective when designing the method was that it should be usable by other people than the designers themselves, notably IVR application developers. Therefore the method has a transparent execution model, and is configurable using a simple scripting language.

1 Introduction

In what follows, we will consider **problem-solving dialogues** with the following characteristics:

- The dialogue participants are a novice and an expert.
- The novice has a problem he cannot solve, but is able to make observations and perform actions.
- The expert has the required domain knowledge to solve the problem, but has a limited capacity to make observations and perform actions.
- Because of this, the novice and expert need to communicate (using natural language) to jointly solve the problem.

Such dialogues appear, for instance, in the context of over-the-phone technical support and troubleshooting. Consider the situation where a service agent is helping to restore a customer's Internet connection. The agent may perform some tests remotely (pinging the customer's computer, checking for network failures, etc), but for the most part the agent tries to nail down the problem by asking the customer to perform a number of actions: restarting the modem, restarting the computer, disconnecting routers and hubs, checking and changing network settings in the computer, etc. The customer mostly acts as an answer supplier and the executor of the actions proposed by the agent.

In this paper, we will consider the challenge of automating the expert by means of a spoken dialogue system. Several issues need to be addressed. First, because the system cannot perform all actions or make all necessary observations, grounding and avoiding misunderstandings become very important. The system must make the user understand what action to perform next, and then itself understand the outcome of that action.

Second, the system must be able to adapt to different users with different levels of domain knowledge. This is particularly important in tech-support domains. While some users are perfectly comfortable with terms like "modem", "command window", "IP number", etc, many others don't know the technical terms, and indeed have very vague conceptions of computers in general. Therefore the system needs to adapt its explanations to the needs of the specific user.

Third, the system must be readily configurable, maintainable, and possible to port to new domains by application developers who do not (need to) know exactly how the system is implemented. To this end, it is important that the system offers a **scripting language** in which applications can be coded. This scripting language must have a transparent execution model, so that developers can foresee all possible situations that can arise during interaction with a user. (This last point is crucial for achieving "VUI completeness" in the sense of Pieraccini and Huerta (2005), and thus a prerequisite for a dialogue system to be useful in an industrial setting).

This paper describes a configurable dialogue manager for problem-solving dialogue applications. It is currently being used in a prototype for providing automated broadband support to the customers of TeliaSonera[1], and we will use examples from this domain throughout the article. An earlier version of the model (not as easily configurable) was used in the "Nice" fairy-tale computer game prototype (Boye and Gustafson 2005, Boye et al. 2006) as a means to control the behavior of virtual game characters (see Sect 8).

2 Problem-solving tasks and dialogues

Consider the Internet connection problem again. The service agent knows that in order for the customer‘s connection to work, several conditions need to be satisfied: the network must be functioning, the user must have paid his bill, and the user's equipment must be functioning and set up appropriately. Put equivalently, if the Internet connection does **not** work, one of the conditions just mentioned must be unfulfilled. This latter formulation suggests a procedure for finding the source of the connection error; just check the prerequisite conditions one at a time, until the error is uncovered. If we find that one condition is fulfilled (e.g. the network is working properly), we can cross it off our check list and proceed to examining the other conditions, as the problem surely must lie with one of them.

Some of the conditions are complex, and can be decomposed further into sub-conditions. For instance, that "the user's equipment is functioning" involves checking that the modem, router, computer, wires, etc. are all ok, and some of these sub-conditions can be further broken down into sub-sub-conditions, and so on. That is, the troubleshooting process takes the form of a **hierarchical task decomposition** process.

The way propositions are expressed in terms of sub-propositions as above is reminiscent of Prolog (Sterling and Shapiro, 1994). The problem could also be viewed as one of finding a sequence of actions that transform the current state of affairs (in which the Internet connection is not working) into a desired goal state (in which the connection works). On this viewpoint, the problem seems amenable to AI planning approaches like STRIPS (Fikes and Nilsson, 1971).

However, both Prolog and STRIPS-like approaches assume that full information is available from the very beginning, and that problem solving amounts to searching through known facts about the domain. This is not the case in the problem-solving domains we are considering. A further complication is the fact that the agent cannot carry out actions directly, and cannot foresee which instructions will be understood or not. This is evident from the following fragment (taken from a longer dialogue):

1. **Agent:** We will need to restart your modem. Do you know where your modem is?
2. **User:** Uh, no.
3. **A:** I'll help you find it. Can you locate the telephone plug in the wall?
4. **U:** Uh, yes.
5. **A:** One of the cables going from the telephone plug should lead to a little box that probably has some lights on it.
6. **U:** Ok.
7. **A:** That is your modem.
8. **U:** Ok, I see it.
9. **A:** Good! Now, the modem has a power cord as well. I want you to unplug that power cord in order to restart the modem. Can you do that now?

The user cannot directly restart his modem since he cannot locate it, so the agent needs to embark on a longer explanation (utterances 3–8).

A striking feature of such troubleshooting dialogues is the amount of effort the agent spends on grounding. In fact, the agent needs to have almost every instruction acknowledged by the user (usually a brief "yes" or "OK" is sufficient). If the user does not know how to carry out an instruction (as in utterance 2 above), or rejects it for some other reason, the agent will either explain further, or abandon the current strategy altogether and try an alternative way to proceed.

[1] TeliaSonera is the leading telecommunications provider in the Nordic-Baltic region in Europe.

Smith and Hipp (1994) proposed the “missing axiom theory” as the driving force in problem-solving dialogue management. In this view, completion of actions is represented by theorems, and making sure that an action has been completed involves constructing a proof for the corresponding theorem. If the proof can not be carried out because some needed axiom is missing, the theorem proving process is suspended, and the user is asked to provide the missing axiom (this amounts to a request to the user to perform an action needed to complete the overall task).

Since Smith’s system, several other researchers have applied hierarchical task decomposition to dialogue, notably Rich and Sidner (1996), Lemon et al (2002), and Bohus and Rudnicky (2003). The approach presented in this paper differs from aforementioned approaches primarily by featuring a much simpler way of scripting dialogue applications. Automated troubleshooting dialogue has recently been addressed by Acomb et al (2007), and by Williams (2007), who uses a statistical dialogue management approach rather than hierarchical task decomposition.

3 Encoding the domain

3.1 Speech acts

An analysis of a corpus of dialogues between human service agents and customers revealed that the vast majority of the agent’s utterances can be described using only six speech acts. These are “**request action**” (e.g. “Locate the telephone plug in the wall”), “**request info**” (“What operating system is your computer running?”), “**request info yes/no**” (“Is your router wireless?”), “**ground status**” (“Now a window should appear”), “**inform**” (“There may be a problem with your router”), and “**acknowledge**” (“Good!”).

After having performed an “**inform**” speech act, the agent is not really expecting any reply from the customer; making an “**inform**” is just granting extra information concerning the state of the troubleshooting process (often used when a topic is introduced (“We will need to disconnect your router”) or when it’s closed (“Now we’ve disconnected your router.”)). In contrast, the “**request info**” speech act requires a reply from the customer, and the agent cannot proceed without it. A **“ground status”** is used when the agent wants to confirm a certain result, for instance that the user can see the “Start” menu appearing on his screen after having clicked the “Start” button. The main purpose of a “ground status” speech act, from the agent’s point of view, is to make sure that the user has indeed carried out and understood the effects of the latest action, and is ready to receive the next instruction.

Similarly, the customers’ utterances can be classified using speech acts such as “**inform”** (typically as a reply to a “request info”), “**inform-yes”** and “**inform-no”** (in response to a “request info yes/no”), “**acknowledge”** (typically signaling understanding in response to a “request action” or “ground status”), “**please clarify”** (signaling non-understanding), and “**please wait”** (when the user needs more time to carry out some action). Additionally, the customer usually states the problem at the very beginning of the dialogue. We will not consider this heterogeneous group of utterances in this article, as they are dealt with using statistical classification methods (see Boye and Wirén 2007 for a description of that system), quite unlike those presented here.

We will also consider instantiations of the basic speech acts. For instance, “locate the telephone plug in the wall” is an instantiation of a “**request action”**, which we will represent as **requestAction(locate_telephone_plug).** In general, we will represent (the semantic value) of an utterance[2] by such an instantiated speech act $\mathbf{f(a_1, a_2, \ldots, a_n)}$, where **f** is the basic speech act, and the arguments $\mathbf{a_1 \ldots a_n}$ are terms representing the additional information. As another example, “the IP number is 131.1.15.23” would be represented as **inform(ipNumber, 131.1.15.23)** whereas “no” would be represented simply as **inform-no.**

We can now encode the entire dialogue example of section 2, as follows:

1. **Agent:** inform(restart_modem);
   requestAction(locate-modem)
2. **User:** pleaseClarify
3. **A:** requestAction(locate-telephone-plug)
4. **U:** acknowledge
5. **A:** requestAction(follow-cable-from-telephone-plug)
6. **U:** acknowledge
7. **A:** groundStatus(locate-modem)

[2] In general, an utterance may be represented by a sequence of speech acts (and not necessarily a single speech act).

8. **U:** acknowledge
9. **A:** acknowledge
   requestAction(unplug-power-cord-from-modem)

3.2 Information state

Relevant information about the domain is stored as attribute-value pairs. For instance, we may conceive of an attribute **ipNumber** whose value is **131.1.15.23**. A "proposition" is any statement of the domain that can be either true or false. In particular, the expression **valueOf(x,y)** denotes the proposition that the attribute x has the value y. Some attributes can only take the values **true**, **false,** or **don't know**. If x is such an attribute, we will take the expression **x** to mean the same thing as **valueOf(x,true)**. For instance, **modem-restarted** means the same thing as **valueOf(modem-restarted, true).** We will refer to the ensemble of attribute-value pairs as the "information state"**.**

A proposition is considered to be true, and stored in the information state, as soon it is accepted by the user. For instance, the proposition **locate-telephone-plug** is added after the user's acknowledgement in utterance 4, and **follow-cable-from-telephone-plug** is added after utterance 6.The proposition **locate-modem** is *not* added after utterance 2 since the user does not acknowledge, but is added after utterance 8. Thus, the presence of a proposition like **locate-modem** in the information state in this case means that the user has confirmed that he has performed the action "locate modem". (One may argue that the user having located his modem is an observation rather than an action. However, the distinction between verified executed actions and verified observations is intentionally blurred.)

Non-Boolean values of attributes are added after an **inform** reply from the user (as, for instance, in the exchange: "What operating system is your computer running?", "Windows"). The presence of the proposition **valueOf(operating-system, windows)** in the information state means that the system has already performed a speech act **request-info(operating-system)**, or obtained the information by some other means. In any case, the question needs not be asked again.

4 Deciding system actions

4.1 Dialogue rules: syntax and informal interpretation

In what follows, we will use a rule-based approach of representing the problem decomposition process outlined previously. A rule for making the user restart his modem might look like this:

```
satisfy(restart-modem) {
  satisfy locate-modem;
  perform requestAction(unplug-power-
cord-from-modem);
  perform requestAction(plug-power-
cord-into-modem);
  perform groundStatus(restart-
modem);
}
```

Informally, such a rule is to be interpreted: "In order to have the modem restarted, first make sure that the modem is located (by the user), then ask the user to unplug the power cord, and then ask the user to plug the power cord back in again. Finally, ask the user to verify that the modem actually has been restarted". (We will return to the formal interpretation of the rule shortly.)

That is, the process of satisfying a certain goal can be broken down into a sequence of steps, each of which is either a sub-goal to be satisfied, an action to be executed, or a condition that should be true. The general form of a rule is

satisfy(G) { B_1; B_2; …; B_n; }

where G is a proposition to be satisfied ("the goal"), and each B_i is an expression of one of the following forms:

- **satisfy** P (where P is a proposition)
- **perform** A (where A is an action, i.e. either a speech act or a request for a non-verbal action, such as pinging the user's computer)
- **holds** P (where P is a proposition)

(We will explain the **holds** construct in end of this section.)

Continuing the example, there are two rules for the sub-goal `locate-modem`, corresponding to two alternative strategies for how the agent can proceed. The simple way of making sure the user has located his modem is simply to ask him:

```
satisfy(locate-modem) {
  perform requestAction(locate-
modem);
}
```

The speech act `requestAction(locate-modem)` could for instance be verbalized as “Do you know where your modem is?”, as in the second sentence of utterance 1 in the example of section 2. If the user okays this request, the system will draw the conclusion that the goal `locate-modem` is fulfilled (i.e. add that proposition to its information state). Another strategy to fulfill the goal `locate-modem` is to give a step-by-step explanation:

```
satisfy(locate-modem) {
    perform requestAction(locate-
telephone-plug);
    perform requestAction(follow-
cable-from-telephone-plug);
    perform groundStatus(locate-
modem)
}
```

This is what the agent does in utterances 3-8 in the example of section 2.

The informal interpretation of the construct “**holds** P” is that the proposition P must be true at that point in order for the rule to be applicable. Usually, it is used as a pre-condition, as in the rule:

```
satisfy(check-network-settings) {
  holds valueOf(operating-system,
windows);
  . . . more . . .
}
```

Unless the system already knows that the user’s operating system is Windows, this rule is not applicable.

We will also allow variables in rules, as in the following rule (variables are prefixed with a “$”):

```
satisfy(valueOf(radio-button($x),$y){
  perform requestAction(tick(radio-
button($x, $y));
}
```

This rule states that one way of ensuring that the alternative $y is ticked in the radio button $x is to ask the user to tick it (whatever the values of $x and $y). The use of variables is a notational convenience that reduces the number of rules by increasing their applicability.

Rules such as these constitute a *static* specification of how the automated agent can go about diagnosing and correcting the error (by “static” we mean that the rules will not change during the course of a dialogue).

4.2 The agenda and the formal interpretation of dialogue rules

During the course of the dialogue, the system makes use of the rules to construct and traverse a **dynamic** tree-structure, the **agenda**, which at any point in time represent current and future goals and actions. The agenda is a tree-structure since goals are represented as parent nodes of the sub-goals and actions needed to fulfill them.

Agenda trees can be defined inductively as follows:

- if P is a proposition, then a single node labeled with “**satisfy** P” is an agenda;
- if A_1 is an agenda, then A_2 is an agenda if A_2 can be constructed from A_1 by means of the following **expansion** operation:
  (1) choose a leaf node L which is labeled “**satisfy** X”
  (2) choose a matching dialogue rule “**satisfy** Y { B_1; … B_n }”, where σ is a binding of the variables in Y, such that σ(Y)=X. Add n children to L, labeled σ(B_1), …, σ(B_n).

As an example, the agendas in figures 1c and 1d (found at the end of the article) are both obtained by expansion (using two different rules) of the agenda in figure 1b, which in its turn is an expansion of the agenda 1a.

Note that it is also possible to transform agenda 1c into 1d by selecting the node labeled “**satisfy** locate-modem”, pruning all children below that node (we will refer to this operation as performing a “cut-off” at that node), and then expanding that same node using another rule.

Whenever the system needs to decide what to do next, it searches, expands and transforms the agenda in order to find the *next action node.* The next action node is always labeled “`perform A`”, where A is taken to be the action to be carried out next.

In order to find the next action node, the agenda is searched depth-first, left-to-right, starting from the top node, ignoring already satisfied goals and executed actions, until the first non-executed action is encountered. More precisely, for each visited node **n**, the following decisions are made:

1. If **n** is labeled "`perform` A":
    a. If A has already been performed (this is determined as described in section 3.2), then proceed to the next sibling node.
    b. If A has not been performed, then **n** is the next action node, and A is the action to carry out next.
2. If **n** is labeled "`satisfy` P":
    a. If P is a true proposition then proceed to the next sibling node.
    b. If P is not true, then proceed to the leftmost child of **n**. If **n** is a leaf node, then expand (using the expansion operation above), and then proceed to the leftmost child of **n**.
3. If **n** is labeled "`holds` P":
    a. If P is a true proposition, then proceed to the next sibling node.
    b. If P is not true, remove **n** and all of **n**'s siblings. Then expand **n**'s parent node, using another rule than before, and proceed to the leftmost child of **n**.

In cases 2b and 3b, the system currently uses the Prolog-like strategy of using the rules in the order they are listed. That is, in case 2b the first matching rule is selected, and in case 3b the first *unused* matching rule is selected.

To illustrate how the system uses the agenda, suppose figure 1a is the starting point. The system would expand the agenda twice, leading to figure 1c. The next action node is thus labeled "**perform** requestAction(locate-modem)", which is what the system will say (verbalized as utterance 1 of the dialogue example of section 2).

Since the user does not acknowledge but rather asks the system to clarify (in utterance 2), the system considers the chosen strategy to be no good. As a reaction, the agenda is rebuilt into figure 1d.

5 Interpreting user input

Each speech act has an associated system utterance, and most of them have an associated grammar. Furthermore, all speech acts have an associated set of **expectations** that tells the system how to interpret the user's input. When a particular speech act is chosen by the system as the next action, the associated utterance is played, and then speech recognition is performed using the associated grammar. If there is no associated grammar, the system assumes that it is its turn to speak again.

After **request action** and **ground status** speech acts, a grammar is used which is capable of recognizing the user speech acts **acknowledge, please clarify,** and **please wait** (speech recognition grammars with semantic attachment rules are used, so there is no need for a separate parsing step). As explained in section 3.2, an **acknowledge**ment from the user makes the system consider the proposition under discussion to be true (and add it to the information state). This is what happens in the utterances 3-8 in the dialogue example. Using the algorithm described in section 4.2, the system traverses the agenda (in figure 1d), and visits the nodes marked A, B, and C, in that order.

On the other hand, if the user asks the system to clarify, the system will abandon its current strategy, and rebuild the agenda. That is what happens after utterance 2, when agenda 1c is rebuilt into agenda 1d. This is done by removing the current action node and all its siblings, and re-expanding the parent node (in this case labeled "**satisfy** locate-modem") using the next applicable rule.

Some speech acts have specially developed associated grammars. For instance, the speech act **requestInfo(ipNumber)** has a grammar recognizing IP numbers, and so on. The recognized utterance will be interpreted as a value for the attribute (**ipNumber**, in this case), unless the user makes a **please clarify** or **please wait** speech act (these are always among the user's options).

6 Associating utterances with tree events

In the algorithm of section 4.2, the agenda is traversed, expanded and transformed in order to find the next action. During this process, a number of **events** are generated, notably

- When a **satisfy** node is expanded (a "**topic intro**" event).
- When a cut-off is performed at a **satisfy** node, and the node is expanded using the next applicable rule (a "**new strategy**" event).

- When a proposition P is first found to be true, after it has previously been found to be false (a "**topic outro**" event).
- When the system attempts to rebuild the tree, but there are no more unused matching rules (a "**cannot solve**" event).

Note that the first two events correspond (roughly) to the "call" and "redo" entry points in the Prolog "procedure box" control flow model (Byrd 1980), whereas the two latter events correspond, respectively, to the "exit" and "fail" points in the same model.

A useful feature in the dialogue manager is that it allows the dialogue designer to associate system utterances to such events. If there is no associated utterance, an event will just pass unnoticed, otherwise the associated system utterance will be generated.

For example, the event **topicIntro(restart-modem)** is generated when the agenda in figure 1a is expanded into figure 1b, and the event **topicIntro(locate-modem)** is generated in the transition from 1b to 1c. Suppose we associate the utterance "We will need to restart your modem" with the former event (and no utterance with the latter event); then this utterance is generated just before the **requestAction(locate-modem)** utterance ("Do you know where your modem is?"). Together, these two make up the system's first utterance in the dialogue example of section 2.

In the same vein, we may associate the utterance "Good!" with the event **topicOutro(locate-modem)**. When the user has finally located his modem (in utterance 8), the proposition **locate-modem** is added to the information state. At that point in time, the agenda looks like figure 1d. When the system traverses it and reaches the "**satisfy**(locate-modem)" node, the **topicOutro(locate-modem)** event is generated just before the system moves to the next node and generates the **requestAction(unplug-power-cord-from-modem)** utterance. Together, these two make up utterance 9 in the dialogue example.

7 Putting it all together

This is a summary of the execution model of the dialogue manager:

1. The agenda is traversed (and possibly expanded or transformed) using the algorithm of section 4.2. All utterances associated with the ensuing tree events are generated.
2. The result of step 1 is an action or a speech act (if there is no result, the dialogue is finished). Perform this action (in the case of a speech act, generate the associated utterance).
3. If the speech act has an associated grammar, perform speech recognition. Then interpret the resulting speech act based on the expectations associated with the system's latest speech act.
4. Go to 1.

8 Other kinds of problem-solving applications

We began the paper by considering dialogues featuring an expert and a novice, trying jointly to solve a problem. The endeavor here has been aiming at automating the expert side of such a dialogue.

Other configurations are also possible. In spoken natural language robot control interfaces, such as considered e.g. in Rayner et al. (2000), the human takes the role of the expert, having the responsibility for long-term planning, whereas the robot is the novice, responsible for executing actions and making observations. If the robot or device has some planning capabilities of its own, the expert-novice distinction is not clear-cut, and plans may be constructed jointly (see Rich and Sidner 1996, Lemon et al 2002).

An interesting situation is when both the expert and the novice are automated. This might be the case in interactive entertainment (Cavazza et al 2002), or in computer games such "Nice" (Boye and Gustafson 2005, Boye et al 2006). The Nice game features two animated characters with whom the user can talk; however they can also communicate with each other and interfere in each other's plans.

The Nice game used the same dialogue management kernel as the one described in this paper. However, free input was allowed (using a stochastic language model for speech recognition, and a separate robust parsing step), and the system was also capable of performing some reference resolution. Another difference is that the tech-support application described here has a fixed overall goal with the dialogue (the top node of the agenda), which is kept throughout. By contrast, the game characters in the Nice game added new goals to the agenda during the dialogue, as a result of the user's requests and questions.

9 Concluding remarks

In the introduction, we stated three important issues: (1) grounding and avoidance of misunderstandings, (2) on-the-fly adaptation to different kinds of users, and (3) ease-of-use for application developers.

Misunderstandings are avoided, or at least made less probable, by not updating the information state without a confirmation from the user. Rules that encode action chains in several steps are best concluded with a **ground status** speech act, which the user has to confirm ("Now you've restarted your modem.", "Ok!").

The system adapts to the user by rejecting the current strategy and replacing it with an alternative strategy (an alternative dialogue rule) as soon as the user indicates that he does not understand. This may amount to no more than replacing a direct request ("Can you restart your modem?") with a more elaborate step-by-step description to achieve the same thing. But it may also mean trying an alternative way to proceed. For instance, if the user is unable to detect the "Start" button on the screen of his Windows computer, the system may instead ask him to press the "Windows" button on his keyboard.

Finally, as concerns ease-of-use for application developers, our initial experiences are positive, though the broadband tech-support prototype is still under development. It is planned to be deployed by the end of 2007.

Acknowledgement: The author would like to thank Mats Wirén and the anonymous reviewers for valuable comments. This work was supported by EU's 6th framework project "COMPANIONS".

References

Acomb, K., Bloom, J., Dayanidhi, K., Hunter, P., Krogh, P., Levin, E. and Pieraccini, R. (2007) Technical support dialog systems: Issues, problems and solutions. *Proc. Naacl'07 Workshop on Bridging the gap: Academic and industrial research in dialog technologies,* Rochester, NY.

Bohus, D., and Rudnicky A. (2003) RavenClaw: Dialog Management Using Hierarchical Task Decomposition and an Expectation Agenda, *Proc. Eurospeech*, Geneva, Switzerland.

Boye, J., and Gustafson, J. (2005) How to do dialogue in a fairy-tale world. *Proc. SIGDIAL.*

Boye, J., Gustafson, J. and Wirén, M. (2006) Robust spoken language understanding in a computer game. *Speech Communication*, 48, pp. 335-353.

Boye J. and Wirén, M. (2007) Multi-slot semantics for natural-language call routing systems. *Proc. Naacl'07 Workshop on Bridging the gap: Academic and industrial research in dialog technologies,* Rochester, NY.

Byrd, L. (1980) Understanding the control flow of Prolog programs, *Proc. Logic Programming Workshop*, Debrecen, Hungary

Cavazza, M., Charles, F. and Mead S. J. (2002) Character-based interactive storytelling. *IEEE Intelligent Systems*, Special issue on AI in Interactive Entertainment, pp. 17-24.

Fikes, R. E., and Nilsson, N. (1971) STRIPS: a new approach to the application of theorem proving to problem solving, *Artificial Intelligence*, 2 (3-4), pp.189-208

Lemon, O., Gruenstein, A. and Peters, S. (2002) Collaborative activities and multi-tasking in dialogue systems. *Traitement Automatique des Langues (TAL)*, special issue on dialogue, 43(2), pp. 131-154

Pieraccini, R., and Huerta, J. (2005) Where do we go from here? Research and commercial spoken dialog systems, *Proc. SIGDIAL*

Rayner M., Hockey B.A. and James, F. (2000) A compact architecture for dialogue management based on scripts and meta-outputs, *Proc. Applied Natural Language Processing (ANLP).*

Rich, C., and Sidner, C. (1996) When agents collaborate with people, *Proc. AGENTS'97, 1st international conference on autonomous agents.*

Smith, R. and Hipp, R. (1994) *Spoken natural language dialog systems: A practical approach*, Oxford University Press.

Sterling, L., and Shapiro, E. (1994) *The art of Prolog,* 2nd edition, The MIT Press.

Williams, J. (2007) Applying POMDPs to dialog systems in the troubleshooting domain. *Proc. Naacl'07 Workshop on Bridging the gap: Academic and industrial research in dialog technologies,* Rochester, NY.

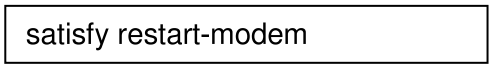

Figure 1(a): An agenda consisting of one node

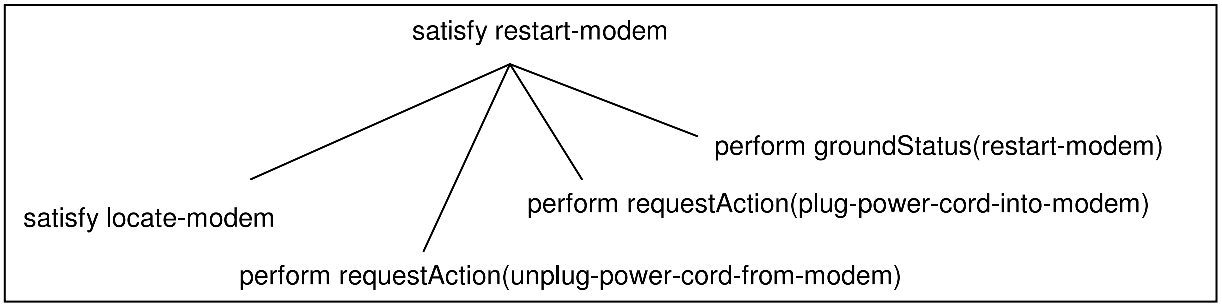
satisfy restart-modem
perform groundStatus(restart-modem)
perform requestAction(plug-power-cord-into-modem)
satisfy locate-modem
perform requestAction(unplug-power-cord-from-modem)

Figure 1(b): An agenda which is an expansion of 1(a)

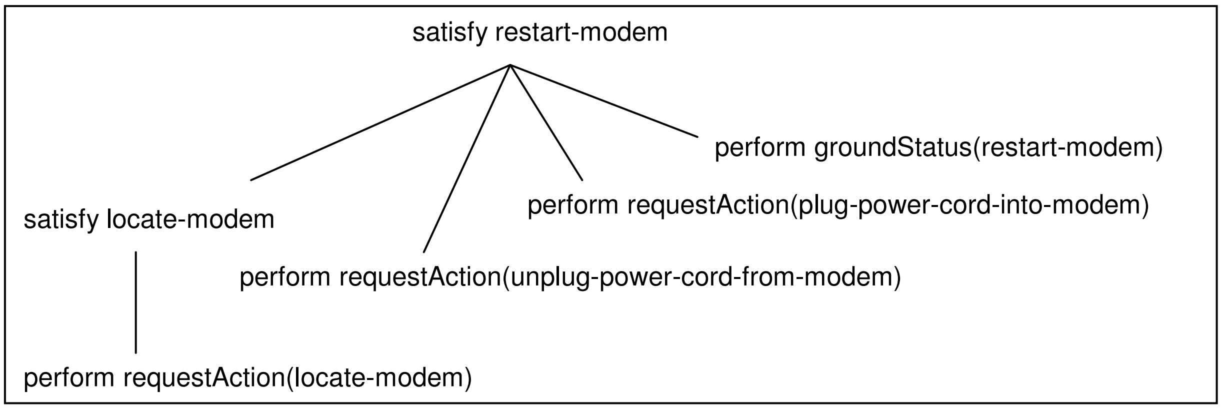
satisfy restart-modem
perform groundStatus(restart-modem)
perform requestAction(plug-power-cord-into-modem)
satisfy locate-modem
perform requestAction(unplug-power-cord-from-modem)
perform requestAction(locate-modem)

Figure 1(c): An agenda which is an expansion of 1(b)

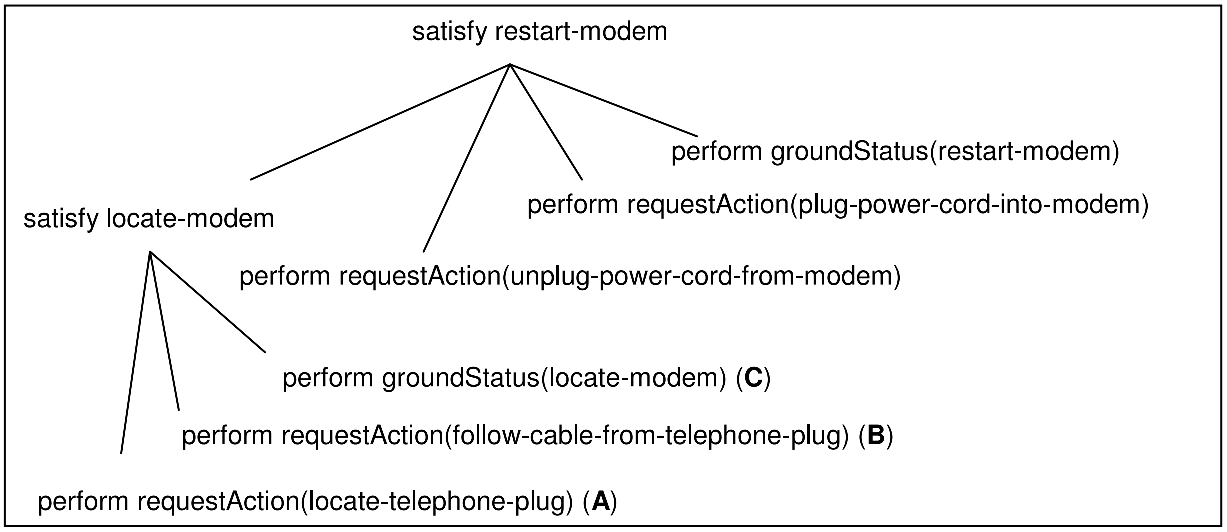
satisfy restart-modem
perform groundStatus(restart-modem)
perform requestAction(plug-power-cord-into-modem)
satisfy locate-modem
perform requestAction(unplug-power-cord-from-modem)
perform groundStatus(locate-modem) (C)
perform requestAction(follow-cable-from-telephone-plug) (B)
perform requestAction(locate-telephone-plug) (A)

Figure 1(d): Another agenda which is an expansion of 1(b)