Asynchronous Dialogue Management: Two Case-Studies

Johan Boye[1], **Beth Ann Hockey**[2], **Manny Rayner**[2,3]

[1] Telia Research
S-123 86 Farsta, Sweden
johan.boye@trab.se

[2]RIACS
Mail Stop 19–39, NASA Ames Research Center
Moffett Field, CA 94035-1000, USA
bahockey@riacs.edu

[3]netdecisions
Wellington House, East Road
Cambridge CB1 1BH, UK
manny.rayner@netdecisions.co.uk

Abstract

Much of the human-machine dialogue research in the literature tacitly assumes a "synchronous" dialogue model; user talks, system acts, system replies. In particular, the user is not supposed to interrupt the system, neither when it talks nor acts. In this paper, we argue that the synchronous model is not appropriate for most interesting real-life applications, but there is a need for asynchronous dialogue models where the user has the possibility to interrupt the system at any time. By referring to two implemented asynchronous dialogue systems, we try to pinpoint what implementation requirements such a dialogue model entails, and we also outline some of the theoretical implications of asynchronous dialogue management.

1. Introduction

The standard model for dialogue management in spoken language interfaces is based on the assumption of turn-taking: user utterances and system utterances will proceed in alternation. In the literature, this assumption is often made so automatically that one isn't even aware of it. Turn-taking typically extends not just to utterances but also to actions. If the system is capable of performing an action (for example, looking up information in a database) in response to a user command, then it is common to assume that the system's turn includes the relevant actions, and that the user will not speak until they are complete. That is, what is often assumed is something we will call the "synchronous dialogue model": user talks, system acts, system replies.

Although this assumption is popular, there are plenty of reasons for doubting that it is in general appropriate. Empirical investigations show that the strict turn-taking model often agrees badly with data from real conversations (Thompson, 1996). As far as practical system-building is concerned, it is now the case that many systems allow at least a limited ability to break the synchronous dialogue convention, and support so-called "barge-in" functionality: the user is allowed to speak before the system has finished talking, breaking it off in mid-stream.

In this paper, we will argue that "barge-in", far from being an isolated exception, is just the most common instance of a range of obviously reasonable dialogue strategies which break the synchronous convention. We will refer to such strategies as "asynchronous". We motivate our arguments with two case-studies drawn from implemented dialogue systems, each of which allows some degree of asynchrony. The basic questions we are asking are the following:

- What practical reasons are there for wanting to support asynchronous dialogue strategies in spoken language dialogue systems?
- What demands does asynchronous dialogue management place on system architecture?

In particular, we have found that the synchronous dialogue model does not account well for situations in which users are talking to a system that takes any appreciable time to do things. For instance, the system might be searching for information over the Internet, or the system might be a robot moving about in the physical world. In these cases, situations will arise where the human wants to interrupt the system not only when it is *talking*, but also when it is *acting*. There are at least two good reasons why this might happen[1]. The first is obvious: the human may not like what the system is doing, and want it to do something else instead. In particular, the human may want the the system to switch to a new behavior, or they may want it temporarily to suspend operations while it executes a new task.

The second reason is slightly more subtle, but becomes apparent as soon as one starts experimenting with a practical spoken language system. Humans get bored easily: even if the system is busy, they would still prefer to talk to it if they can find something useful to say. For example,

[1]A third, which we will not discuss further, is that the human may not even be aware that they are interrupting.

they may want to give the system a new task that can be executed in parallel with the current one, or after it.

The two points meet up when we consider how to organize confirmation strategies. There is generally a tension here between two conflicting goals; telling the user what you intend to do decreases the risk of a misunderstanding, but carries the penalty of slowing down the dialogue and consequently the execution of the task. An asynchronous dialogue architecture allows the possibility of a compromise, since the system can start executing the task and tell the user what it is going to do simultaneously. As long as choosing the wrong task is not actively dangerous, this tends to be a good way to operate: the user relies on being able to interrupt the system if necessary and correct it, but loses no time if the system understood correctly.

Conversely, the system may want to interrupt the human. Something may come up which it considers more important than its current task; alternately, it may be doing several things at once, and need to keep the user informed about their relative progress.

The rest of the paper is structured as follows. Section 2 describes our two sample applications, focusing on the question of how they realize some of the functionalities discussed above. Section 3 relates our findings to other established frameworks for dialogue management and concludes.

2. Two case studies

In this section, we present two case studies of systems that use asynchronous dialogue management strategies. SMARTSPEAK (Boye et al., 1999) is a travel planning system that fetches information from a web-server. Since this takes appreciable time (typically around 20 seconds to a minute), people want to be able to talk to the system while they are waiting for the web-server to return. This means that in general the system is in the middle of a new conversation by the time the web-server gets back.

The Personal Satellite Assistant PSA (PSA, 1999; Rayner et al., 2000) is a simulated version of a semi-autonomous speech-enabled robot intended for deployment on the International Space Station. The robot acts as a mobile sensor: it can go to different places and measure status variables such as temperature and pressure. Since the robot, once again, takes non-trivial time to carry out commands, the possibility arises that the user may want to interrupt them. We now describe the architectures of these two systems.

2.1. SMARTSPEAK

The architecture of the SMARTSPEAK system is based on having a set of independent modules (or agents) communicate asynchronously by message passing. In particular this entails that the agents have no a priori decided execution order; an agent starts executing when it receives a message, and as soon as it has finished executing it is ready to process the next message, regardless of the current state of the other agents. Hence in principle all the agents could run in parallel in different processes.[2]

The Dialogue Manager (DM) is the heart of the system. It can *receive* messages from the language analysis agents (parsed utterances), and from the database agent (database query results). The DM can *send* messages to the speech synthesizer (system utterances) and to the database agent (database queries). The DM maintains a *dialogue state*, which is transformed as a result to each incoming message. How the DM uses its dialogue state to interpret user utterances, resolve references, select system utterances, etc., is described in Boye et al. (1999).

The database agent (DA) receives messages (database queries) from the DM. The DA first checks its local state to see whether the query has been processed before, so that the results are cached. If so, the DA immediately sends a message back to the DM containing the results. If not, the DA sends a message to the DM indicating that the query will take some time to process, and then spawns a process that contacts the travel database web server via the Internet. When the search process returns its results to the DA, the DA will send the results in a message to the DM.

The asynchronous communication between the DM and the DA has several implications, most notably that it is possible that search results do not return to the DM in the same order the queries were sent (e.g. when the results of the second query were cached, but the results of the first query were not). This of course creates complications for the DM, but on the other hand the system can cope with dialogues like the following:

(a) U: I want to go from Stockholm to Gothenburg on Monday morning.

(b) S: I'm searching the database – it will take about 30 seconds. Do you want a single or a return trip?

(c) U: A return trip.

(d) S: When do you want to go from Gothenburg to Stockholm?

(e) U: On Tuesday afternoon.

(f) S: I have received information about trips from Stockholm to Gothenburg. There is a train at ...

(g) U: Fine, I want to book that please.

(h) S: I have booked a train on Monday at ... I have received information about trips from Gothenburg to Stockholm. There is a train at ...

As concerns the communication between the agents, the following points are worth noting.

1. The user utterance (a) makes the DM send off a search query to the DA.
2. The DA sends back a message that the search will take some time, which triggers the DM to generate the first sentence of utterance (b). The DM then initiates a conversation about a second topic (the return trip).

[2]In the current implementation there are four constantly running processes, which handle speech recognition, speech synthesis, database access, and everything else, respectively (where "everything else" essentially means language analysis and dialogue management). In addition the database agent will spawn short-lived processes; see the main text.

3. The user utterance (e) makes the DM send off a second query to the DA.

4. At some point between (b) and (f), the results concerning the outbound trip are sent from the DA to the DM. (The DM will not present the result as soon as they come in, but will wait until it thinks the moment is right.)

5. At some point between (e) and (h), the results concerning the return trip are sent from the DA to the DM.

Figure 1 shows the communication between the different agents in graphical form.

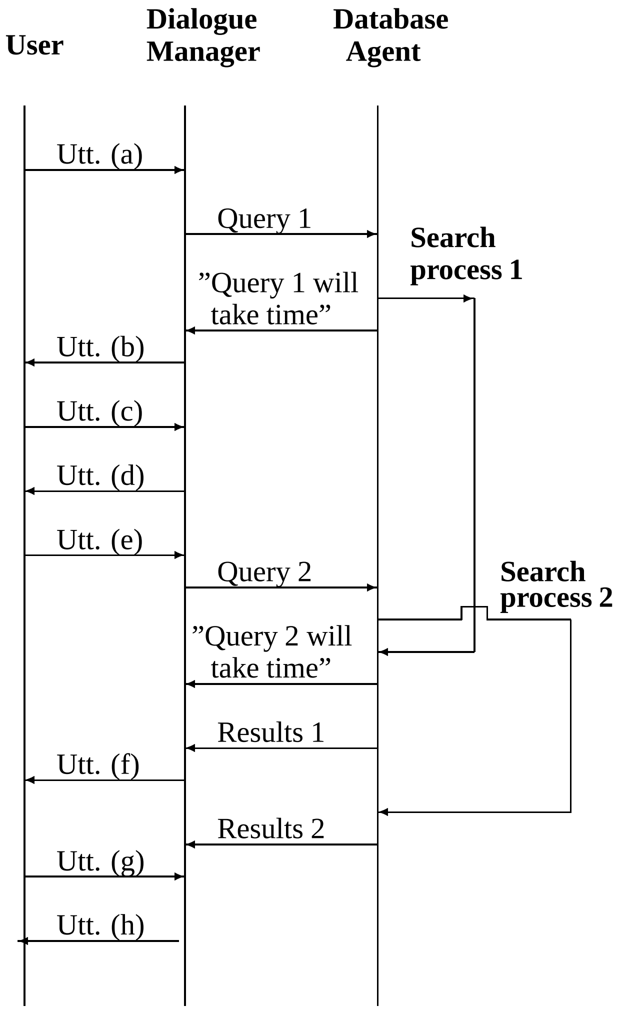

Figure 1: Agent communication in the Smartspeak example

2.2. PSA

The PSA system is configured as a set of independent agents connected using the SRI Open Agent Architecture OAA (Martin et al., 1998). In accordance with the usual OAA design philosophy, each agent is an independent process which maintains its own state. Agents communicate by means of calls in Interagent Communication Language (ICL), an extension of Prolog. Calls can be either synchronous (execution of the calling agent suspends until the call is complete, returning a value) or asynchronous (execution of the calling process continues, and no value is necessarily returned). The agents that are of interest to us here are the following:[3]

Speech recognition and parsing (SRP) The agent attempts to recognize input speech, and if successful produces an output semantic representation, which is sent as an asynchronous message to the dialogue manager agent.

Dialogue manager (DM) The dialogue manager is the central agent: it receives semantic representation messages from the SRP, and decides on the next dialogue action. There are a number of possibilities, of which the most important are the following. (1) Convert the semantic representation into an executable form (a "script"), and pass it to the Action Manager as an asynchronous message; (2) Send a request to the Action Manager specifying a modification to the executing script; (3) Send a request to the Generation agent to create a confirmation question (this can be combined with (1)), a clarification question, or some other verbal response.

The Dialogue Manager's behavior is explained in detail in Rayner et al. (2000).

Action manager (AM) The Action Manager consists of two interrelated pieces of functionality. The Executive subsystem receives scripts from the DM and executes them. Scripts are complex structures composed of atomic actions; execution is ultimately performed by sending synchronous atomic action messages to ROBOT.

The Action Manager also includes a second subsystem, the Interrupt Blackboard, which is used to hold state relevant to processing of interrupts and related requests. Other agents can write to and read from the Interrupt Blackboard at any time, irrespective of whether the Executive is busy.

Simulated PSA Agent (ROBOT) The back-end simulated PSA application, which executes the action messages produced by the AM.

Generation and speech synthesis (GSS) Turns abstract representations of utterances into speech.

The following simple dialogue illustrates how processing works.

(a) U: Go to flight deck.

(b) S: I am going to flight deck. *[Simultaneously starts moving towards flight deck]*

(c) U: Stop. *[Robot stops]*

(d) U: Measure temperature.

[3]In the interests of expositional clarity, we have omitted some agents and collapsed others into single agents. In particular, the Action Manager, which conceptually forms a single unit, is concretely realized as two OAA agents.

(e) S: The temperature at the current location is 19 degrees Celsius.

(f) U: Continue. *[Robot resumes moving to flight deck]*

In terms of message traffic between agents, the critical points are the following.

1. The user says (a). This causes SRP to send an asynchronous message to the DM, which in turn sends messages to AM (the script to execute) and GSS (the confirmation question to ask). Both messages are asynchronous, so SRP and DM are free to process new messages while AM and GSS are active.

2. The message sent to GSS causes it to say (b). AM sends a synchronous message to ROBOT to start the simulated move command.

3. When (c) is uttered, DM updates the state of AM's Interrupt Blackboard to ask for an interrupt. ROBOT, which is continually monitoring this blackboard, abandons processing of its current command and returns control to AM. Since the Blackboard has a stop message posted, AM pauses and waits for further instructions.

4. The user utters (d). DM now sends a new script to AM. Since AM is in a stopped state, it recursively invokes a new copy of the command interpretation loop from the interrupted point in the current execution. This executes the script (causing GSS to say (e)), and on completion returns control to the previous stopped state.

5. When the user utters (f), DM changes the state ofAM's Interrupt Blackboard from "stop" to "resume". This causes AM to continue from the stopped execution state.

Figure 2 presents the processing described above in graphical form; in order to highlight the correspondences with Figure 1, we have hidden SRP and GSS, which do no more than relay messages between the user and the other agents.

In the next section, we compare the architectures of the SMARTSPEAK systems and PSA systems with others described in the literature, and attempt to draw some general conclusions.

3. Conclusions: Architectures for Asynchronous Dialogue Management

To sum up, asynchronous dialogue management allows for the construction of spoken dialogue system that can *act* and *talk* at the same time. In particular, such systems are able to execute a command from the user, and at the same time receive the user's next command. We want to emphasize that this is not just a cute twist or a question of efficiency, but is rather a *fundamental requirement* when equipping real-time systems (like robots) with a spoken dialogue user interface. In many such applications, it will be unavoidable that the system is busy doing something when the user talks to it; hence for such applications the "synchronous" dialogue model tacitly assumed in much dialogue research is helplessly inappropriate.

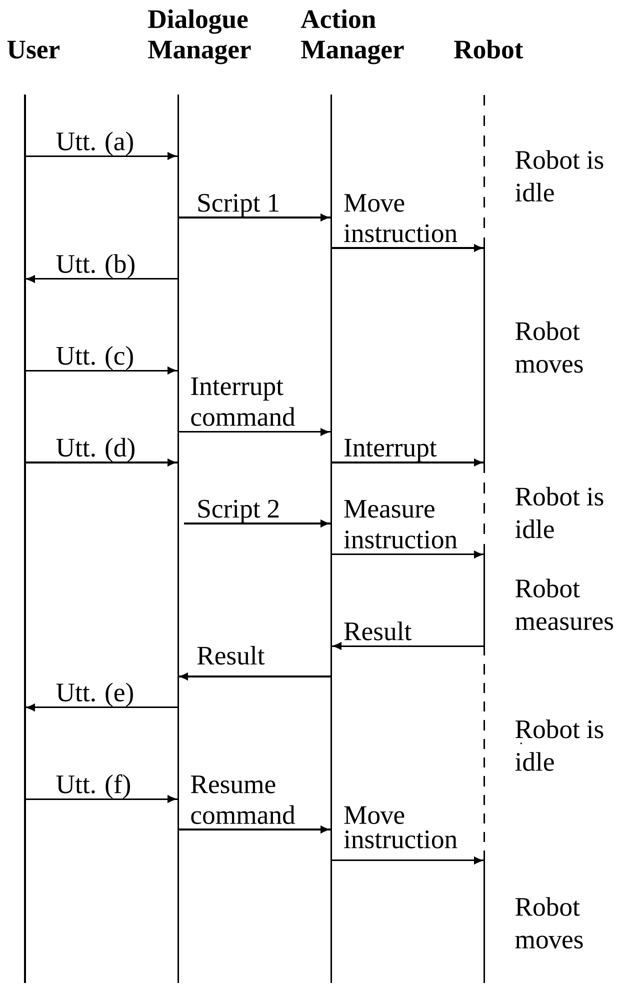

Figure 2: Agent communication in the PSA example

Furthermore, although we have emphasized the practical implementation issues in this paper, asynchronous dialogue management also poses some difficult theoretical problems which we have barely touched upon here (and to the best of our knowledge, have received little attention elsewhere). For instance:

- How should the system deal with a user command ordering the system to do action A, when it is currently busy performing action B? There are a number of possibilities: the system may execute A and B *in parallel*, or it may execute B after A is finished (*sequential* execution), or it may *abort* B, or *suspend* B (in order to resume B after A has finished executing). Or, finally, the system may choose to ignore the user's command altogether. The question is under which circumstances which alternative is preferable, and what criteria the system can use to decide the appropriate course of action.

- Assuming that there are several ongoing and/or suspended activities, how does the system determine the appropriate context in which to interpret the user's utterances, what are the preferred reference resolution strategies, etc.?

- How does the system schedule its utterances, so that the resulting dialogue is comprehensible for the user?

Although a few researchers pay lip-service to the concept of asynchronous dialogue management or handle some special cases, there seems to be surprisingly little acknowledgement that it is important. This is highlighted by the influential DARPA Communicator project DAR (1999), which is currently being used by a large number of research site in the US and Europe. The current Communicator architecture derives from GALAXY-II (Seneff et al., 1998), and organizes the system as a number of stateless "server" processes, controlled by a "script" run from a central "hub" process. This architecture is not aimed towards asynchronous communication between processes. For example, Aberdeen et al. (1999) contrasts Communicator with OAA as follows:

> Both schemes provide for flexible flow of control. However while flow of control is explicitly programmed in the Hub, in the OAA it is determined autonomously by interactions between the agents . . . In sum, the Communicator allows for programmable but pre-determined flows of control while the OAA allows for dynamic but not pre-determined flows of control.

We are not claiming that the examples in sections 2.1. and 2.2. are startlingly novel or complex. The point we want to make is that being able to deal cleanly with this kind of thing makes certain demands on the architecture of a spoken language dialogue system. The processing is relatively simple because we have multiple asynchronously acting agents, each of which has independent state and is able to make requests of the other agents. In particular, we have separate agents which contain *dialogue state* and *action state* respectively. Each of these types of state constitutes a context which needs to be maintained, and which is essential to the interpretation of commands.

Although it would be possible to implement similar functionality using a centralized architecture like Communicator, this would be much less straight-forward: in particular, we would have to reify action state as an object which could be passed between the server taking the role of the Action Manager and the Hub. One could do so, but we feel that this is really somewhat beside the point. Our basic argument is that dialogue management is best conceptualized as a distributed and asynchronous process; if we are prepared to grant this, it certainly seems natural to conclude that it will be easiest to represent it in a distributed and asynchronous form.

4. References

J. Aberdeen, S. Bayer, S. Caskey, L. Damianos, A. Goldschen, L. Hirschman, D. Loehr, and H. Trappe. 1999. Implementing practical dialogue systems with the DARPA Communicator architecture. In *Proceedings of the IJCAI'99 Workshop on Knowledge And Reasoning In Practical Dialogue Systems*, pages 81–88.

J. Boye, M. Wirén, M. Rayner, I. Lewis, D. Carter, and R. Becket. 1999. Language-processing strategies for mixed-initiative dialogues. In *Proceedings of the IJCAI'99 Workshop on Knowledge And Reasoning In Practical Dialogue Systems*, pages 17–24.

1999. *DARPA Communicator Web Page*. http://fofoca.mitre.org. As of 14 February 1999.

D. Martin, A. Cheyer, and D. Moran. 1998. Building distributed software systems with the open agent architecture. In *Proceedings of the Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, Blackpool, Lancashire, UK.

1999. *Personal Satellite Assistant (PSA) Project*. http://ic.arc.nasa.gov/ic/psa/. As of 14 February 1999.

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

S. Seneff, E. Hurley, C. Pao, P. Schmid, and V. Zue. 1998. Galaxy-II: A reference architecture for conversational system development. In *Proceedings of the 5th International Conference on Spoken Language Processing*, Sydney, Australia.

H. S. Thompson. 1996. Why 'turn-taking' is the wrong way to analyse dialogue: Empirical and theoretical flaws. In *Proceeding of the 1996 International Symposium on Spoken Dialogue*.