The Senior Companion Multiagent Dialogue System

(SHORT PAPER)

Hugo Pinto
hugo@hugopinto.net
University of Sheffield

Yorick Wilks
Y.Wilks@dcs.shef.ac.uk
University of Sheffield

Roberta Catizone
R.Catizone@dcs.shef.ac.uk
University of Sheffield

Alexiei Dingli
A.Dingli@dcs.shef.ac.uk
University of Sheffield

ABSTRACT

This article presents a multi-agent dialogue system. We show how a collection of relatively simple agents is able to treat complex dialogue phenomena and deal successfully with different deployment configurations. We analyze our system regarding robustness and scalability. We show that it degrades gracefully under different failures and that the architecture allows one to easily add new modalities was well as porting the system to different applications and platforms.

General Terms

Algorithms, Design, Reliability.

Keywords

dialogue, natural language processing, agent-oriented software engineering, emergent behavior, scalability, robustness, performance, case studies.

1. THE SENIOR COMPANION

The Senior Companion is an application designed to make company to senior citizens, help them carry out their everyday tasks and provide them with easy access to information, including past conversations. It is intended to be deployed in a variety of devices, from computer desktops to handheld devices and small robots.

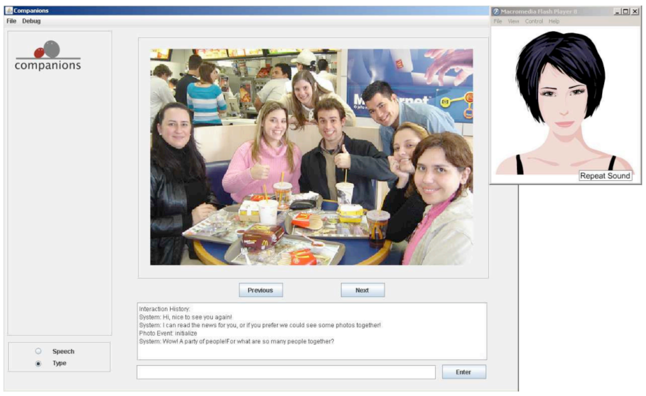
The current implementation builds a user model by carrying out conversations over a collection of photos. A photo, besides being the direct object of some conversations, may act as a trigger to conversations about the relations of the user with the people, places and events depicted.

Regarding everyday tasks, the current system it is able to read daily news from selected websites to the user.

The user interacts with the system mainly via voice or text, if it is plugged to a microphone or keyboard-enabled device, respectively. In its most traditional deployment, a touch-screen laptop, the user can also use touch for a limited set of commands (an example being selecting a person in the photo).

Figure 1 shows the view of the system deployed in a standard desktop computer that has a microphone. On the right side one notices an avatar. Lip movements, head movements and facial expressions associated with the communicative act are displayed through it. In the center, the system displays the photo that is the subject of discussion or an icon representing the current activity of the system. For instance, a newspaper is shown if it is currently reading the news. The buttons allow the user to go to the next or previous photo. In the lower part there are two text panels, the uppermost shows the history of the user interaction so far, and the one in the bottom takes typed input.

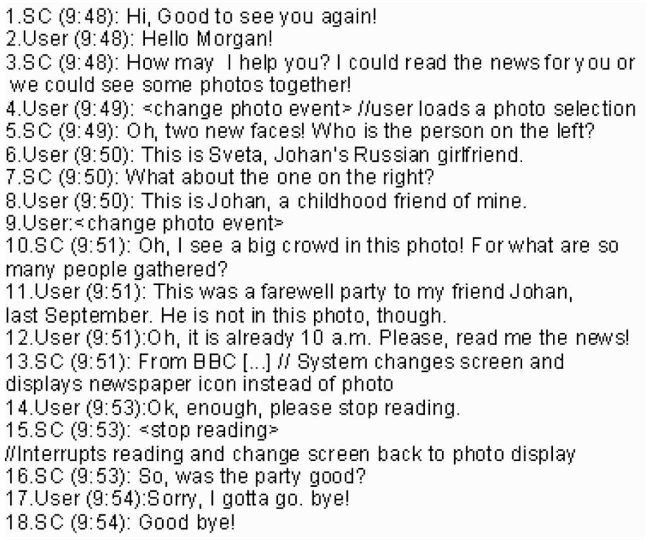
Cite as: The Senior Companion Multiagent Dialogue System (Short Paper), H. Pinto et al., Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 1245-1248.



```
Interaction History:
System: Hi, nice to see you again!
System: I can read the news for you, or if you prefer we could see some photos together!
Photo Event: initialize
System: Wow! A party of people!For what are so many people together?
```

Figure 1: Senior Companion Screenshot

Figure 2 shows the annotated log of a typical user session. SC stands for senior companion. Comments after // describe the action carried out by the user or the system that does not appear in the log.



```
1.SC (9:48): Hi, Good to see you again!
2.User (9:48): Hello Morgan!
3.SC (9:48): How may  I help you? I could read the news for you or
 we could see some photos together!
4.User (9:49): <change photo event> //user loads a photo selection
5.SC (9:49): Oh, two new faces! Who is the person on the left?
6.User (9:50): This is Sveta, Johan's Russian girlfriend.
7.SC (9:50): What about the one on the right?
8.User (9:50): This is Johan, a childhood friend of mine.
9.User:<change photo event>
10.SC (9:51): Oh, I see a big crowd in this photo! For what are so
many people gathered?
11.User (9:51): This was a farewell party to my friend Johan,
last September. He is not in this photo, though.
12.User (9:51):Oh, it is already 10 a.m. Please, read me the news!
13.SC (9:51): From BBC [...] // System changes screen and
displays newspaper icon instead of photo
14.User (9:53):Ok, enough, please stop reading.
15.SC (9:53): <stop reading>
//Interrupts reading and change screen back to photo display
16.SC (9:53): So, was the party good?
17.User (9:54):Sorry, I gotta go. bye!
18.SC (9:54): Good bye!
```

Figure 2: Annotated Log of a User Session

2. THE MULTIAGENT DIALOG SYSTEM

Our dialog system is composed of a collection of agents.

The general protocol for inter-agent communication is a simple type-based publish-subscribe mechanism, similar to the one used in SmartKom[2]. Each listener agent subscribes with the agents that publish the types that it listens for, and each producer agent broadcasts changes to its listenable types to the subscribed agents. Any agent can be at the same time a subscriber and a publisher.

Figure 3 shows the main agents and components of the senior companion.

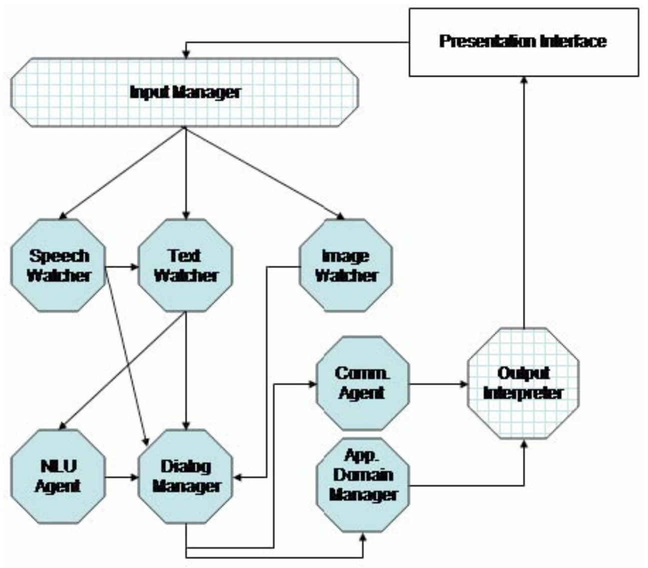


Figure 3: Senior Companion Agents and Modules

The *Input Manager* translates the inputs from the user interface to messages that are understood by the dialog system agents. The Text Watcher, the Speech Watcher and the Photo Watcher listen to the availability of new voice, text or photo respectively.

The Speech Watcher processes speech (a wave file) and publishes two results: a string corresponding to its highest scoring hypothesis and a table with its N-best hypotheses and their corresponding texts.

The *Text Watcher* agent listens for the presence of a new text string that might come from the Input Manager or the ASR Agent, and broadcasts a potentially modified version of the received text. It might apply simple corrections in the input before broadcasting, such as changing "waht" to "what".

The *Image Watcher* detects whenever a new photo event comes through the input interface and publishes the information related to the photo (GPS coordinates, time, number of people detected, etc). It uses a customized version of OpenCV for image processing[].

The Natural Language Understanding (NLU) Agent processes a text string and outputs a linguistic interpretation of the sentences present. It includes a syntactic analysis and a shallow semantic analysis that includes named entities. This agent uses a customized version of GATE's ANNIE system[1] for the analyses.

The *Dialogue Manager* (DM) uses information about the current interaction, the interaction history, its background knowledge and the information from the NLU Agent and the watchers to decide what to do at each time step. The DM's decision of what to do at each time step may be to issue a command (perform<action>), to convey a dialogue act (*convey<dialog-act>)* or to just wait. As the dialog manager itself is a complex multiagent system, we will defer its details to the next section.

The *Application Domain Manager* (ADM) decides how to satisfy the logical specification of commands from the *Dialogue Manager* (DM) with a set of domain-specific commands. We have one ADM per application domain.

The *Communication Agent* takes a *convey(DialogAct)* command and decides how to present the information. It is up to this agent to decide if it will use the gesture of an avatar, voice, the application canvas or a combination of both to display the information. For instance, it could decide to present *convey(inform(movies_info))* as a table showing the cinemas and movies, if *movies_info* had entries for several movies, or it could just render it as speech if *movies_info* had a single entry.

The *Output Interpreter* (OI) agent takes a logical specification of a list of commands and translates it into the actual commands understood by the interface. For example, it might render a specified table as HTML, vector graphics or GIF image, depending on the capabilities of the GUI. We have one output interpreter agent for each specific user interface and GUI. As an example, we could have one for Internet Explorer and another for Firefox.

The *Presentation Interface* is responsible for playing the sounds, actually loading and changing pictures and gathering user input.

3. DIALOGUE MANAGEMENT

The dialog manager, though an agent from a higher level viewpoint, is also a multiagent system composed of two main agent types: behavior and control agents.

Behavior Agents embody a single conversational or operational task of the system, such as "discover user name", "chat_about_photo", "talk about event" and "read news".

Control agents are used to determine which behavior agent will run at each time step.

Each behavior is tagged at design time with a set of terms that characterize it. These terms can be restrictions on domain properties (such as time>18:00), keywords, parts-of-speech tags, named entities, or syntactic dependencies. Each behavior in the system has a unique key, formed by the set of its terms.

The dialog manager uses the information that comes from the watchers and the NLU agent to make a set of indexing terms. It matches these indexing terms to the keys of each behavior and selects for execution at each time step the behavior that most closely matched the current indexing terms.

But what happens if the user interrupts a conversation in its middle? How may the system get back to it later?

We use a simple solution for this dilemma: we put the behavior into a stack. At any moment the running behavior is the one that is on the top of the stack. Whenever a new conversational behavior is selected, the current one is stopped and the new one pushed on top of the interrupted behavior. The dialog proceeds according to the new behavior. When it is over, the corresponding behavior is popped and the interrupted behavior and conversation resumed from where they were stopped.

A problem arises with this approach: what to do when the user causes the selection of a behavior that is already pushed down the stack? In most cases we do not want to restart the conversation from scratch, but rather, we want to continue the conversation from were we left. We thus made the system stack a "white-box stack with removal', allowing the inspection and removal of elements in any position of the stack.

Figure 4 shows the Dialog Management System. Dashed arrows indicate the direction of broadcast messages. Full arrows indicate components that directly modify others. Full connections that are not arrows show just association.

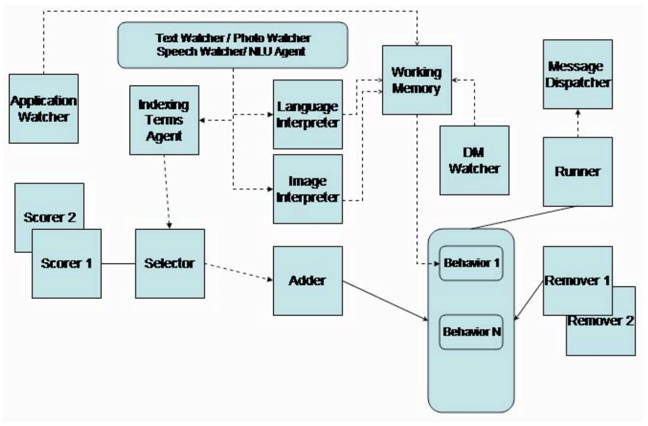


Figure 4: Agents of the Senior Companion Dialog Manager

The *Adder* is the agent that takes care of the decision of what to do when a new behavior is selected for execution, as discussed in the previous paragraph. In this system it checks the stack for an interrupted behavior identical to the one to be stacked, and if it is the case, remove it from the stack and re-pushes it to the top. Otherwise it just pushes the new behavior. A behavior, when selected for execution, keep the values of the indexing terms used for its selection – it is then called an instantiated behavior. These terms provide a partial context for the behavior.

The *Remover* takes care of a problem that we have not discussed so far: how to perceive and decide when a conversational behavior is no longer relevant and what to do when it happens? Examples could be conversations that were interrupted for so long that the user is no longer interested in them, or conversations that were subsumed by other conversations. The current system has two behavior *Remover* agents, one that removes behaviors that are inactive for a time longer than a constant and another that removes behaviors that get pushed down the stack beyond a certain depth.

The *Working Memory* (WM) is not as passive as the name might suggest. Besides keeping predicates and objects that correspond to the knowledge of interest to the behavior agents, is has three active roles: it tries to keep its knowledge consistent, actively forgets old information, and automatically infers new information whenever new predicates and objects are added to it. As the other agents in the system, it notifies subscribed agents of changes in its state. The *Adder* registers the instantiate behavior to listen to events in the *Working Memory*, so that the behavior always have the latest percepts during its execution.

The low level percepts coming from the *Text Watcher*, *Photo Watcher*, *Speech Watcher* and *NLU Agent* are caught by the *Indexing Terms Agent*, the *Language Interpreter* and the *Image Interpreter*.

The *Language Interpreter* adds predicates and objects to the *Working Memory*, based on what is already there, the system background knowledge and the outputs of the *NLU Agent*, *Speech Watcher* and *Text Watcher*. It is the element that might be able to tell that "he', corresponds to George_Washington_01, a specific George Washington in our system.

The *Image Interpreter* uses information from the watchers to populate the working memory in the same way as the Language Interpreter. One example is the addition of predicates that describe the relative positions of the people described in the photos. It has background knowledge about pictures and spatial relations.

The *Application Watcher* is an application and system specific agent that creates objects and predicates representing aspects of the system that are used in behavior selection and execution. It is the agent that might populate the working memory with system time information, for example.

The *Indexing Terms* agent uses the information of the WM, the NLU, and the watchers to create indexing terms for behavior selection.

A *Scorer* agent, under request of a *Selector*, produces a list of scores, each corresponding to a particular view of the indexing terms of the behaviors available for selection. We may have scorers that focus only on full matches, scorers that use term expansion to assign partial scores, scorers that just consider system properties, etc. The main motivation for this was to allow experimentation with different scoring policies, and to being able to treat each term type individually. The present system uses just full match scorers, one for system properties and one for keywords.

The *Selector* agent decides which behavior, if any, will be selected for addition whenever it receives new indexing terms. It calls the available scorers and uses a defined algorithm to combine them. Currently we select the highest scoring behavior considering the sum of all *Scorers*. In the future we will investigate the incorporation of default preferences and preferences based on the content of the stack.

The *Dialog Manager Watcher* (DM Watcher) monitors the events inside the dialog manager. It populates the Working Memory with predicates such as "NewUtteranceArrived(time)". The predicates and objects of the Dialog Manager are used in operations of finer grained dialog control and repair, usually carried out by specialized behaviors (an example would be "clarify last question").

A behavior in our system is ultimately implemented by an augmented finite-state machine(more specifically an augmented transition network[7]). Any action or check is performed by sending a message and receiving an acknowledgement (the FSM may ignore the acknowledgement, if it is not crucial)

The *Behavior Runner* (BR) is the agent that actually drives behavior execution, telling a stacked behavior when to be active and when to wait. It won't stop a behavior in the middle of a transition or action though, so the behavior always stops immediately after performing a transition or immediately before checking the transition conditions. The *Behavior Runner* is also the agent that removes behaviors that have finished their execution.

Finally, the *Message Dispatcher* is the agent that processes the messages from a behavior. It publishes the dialog system messages in a form amenable to the *Communication Agent* and *Application Domain Manager*.

4. DISCUSSION

In this section we discuss our system regarding scalability and robustness, compare our system to related work and point to our future investigations.

4.1 Robustness, Scalability and Configurability

Our main motivation to adopt a multiagent approach to our dialog system was to be able to easily configure, modify, adapt and monitor it, was well as making it robust. These characteristics are particularly relevant as the requirements and deployments of our system might change beyond our predictions, and the set of linguistic and cognitive hypothesis to explore are large.

One question of central interest, considering verbal expression, is how the various scoring and selection strategies affect the overall dialogue behavior. The same applies to the behavior removal policy – how the removal strategy and frequency might impact dialogue? Could we use it to create a ”obsessive” system, that kept going back to everything the user said, even if it happened three hours before? What about the different ways to add a behavior?

We see that by decoupling scoring, selection, addition and removal of behaviors we create not a single system, but a family of systems that can be easily realized by different agent compositions. We used in our default implementation agents that embody strategies that embody one particular successful deployment[5], but we intend to explore the spectrum of plausible combinations in the future.

The dialog management system may continue to function, albeit degraded, even in the absence or failure of some key inputs or agent. For instance, if we have no NLU input, the selection and scoring of behaviors will be carried out considering just the words of a sentence. If speech fails, the user will still be able to use the system using typed input. If one *Scorer* does not return a valid result but the others do, the *Selector* will still be able to do an informed decision.

If we add a whole new modality to our system, such as gestures, we need to add just a gesture Watcher and a Gesture Interpreter, without any need to modify the rest of the system (except the concerned behaviors).

We have integrated our dialogue system with a web-based interface that allowed the user to interact with our system using a browser. This web interface had an embedded avatar. We also integrated it with a standalone Java application that had no avatar, and later superimposed another application containing an animated avatar (shown in Figure 1). We were able to carry out each integration with minimal effort – we had to modify just the Input Manager, the Output Interpreter in all cases. Between the first and second integration we had to build a new Communication Agent and a new Application Domain Manager. When we added the third application, we had to modify the Communication Agent to deal with the extended capabilities of the animated avatar, but the ADM was unchanged as the basic application functionality remained the same. This made us confident in the suitability of the system to be deployed at different devices and its robustness and portability to different scenarios.

4.2 Related Work

The use of ATN's and a Stack for language processing go back at least to Woods[7], albeit in a syntax context. For dialogue, this combination was applied in the COMIC and mini-CONVERSE[5] projects. The main difference from our approach is that the behavior selection system was monolithic and the behaviors (called Dialog Action Forms) directly controlled the application and the system. Contrary to our system, the behavior could not be ported if we changed the application, and any change in selection or scoring would cascade over many parts of the entire system.

The idea of combining variations of finite-state-machines and a Stack to be able to interrupt and resume agent behaviors was also extensively used to control agents in computer games[4].

Inspiration for the actual software engineering of our behaviors came from [6].

Our scoring system has a mild resemblance to the general functioning of the multiagent system in the Jaspis[3] architecture, in that we decouple scoring and selection of an agent that embodies the actual behavior of a task. However our behaviors do not provide a default score as in Jaspis and we used this organization just or a specific case.

5. ACKNOWLEDGMENTS

Wei Wei Cheng was a co-designer and main implementer of the interface shown in Figure 1, and also implemented the interface-side features of the Input Manager and Output Interpreter. The Avatar shown in Figure 1 was built by As An Angel and modified to interoperate with the Senior Companion by Benoit Tabutiaux. We thank Thiago Tamosauskas for providing us the pictures used in the system's tests. This work was supported by the EC.

6. REFERENCES

[1] Cunningham, H., D. Maynard, et al. Framework and Graphical Development Environment for Robust NLP. Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02). Philadelphia. (2002).

[2] Wahlster, W. (Ed). SmartKom: Foundations of Multimodal Dialogue Systems. (2006).Springer-Verlag.

[3] Turunen, M et al. An Architecture and applications for Speech-based accessibility systems. IBM Systems Journal Vol 44, N 3, (2005).

[4] Houlette, R. and Fu, D. "The Ultimate Guide to FSMs in Games". *AI Game Programming Wisdom 2*, Steve Rabin(Ed). 2003.

[5] Catizone, R., Setzer, A., and Wilks, Y. "Multimodal Dialogue Management in the COMIC Project", Workshop on Dialogue Systems: interaction, adaptation and styles of management. (EACL)Budapest, Hungary. (2003)

[6] Guessoum, Z.,Briot, J.-P. From Active Objects to Autonomous Agents. IEEE Concurrency 7, 3 (1999) 68-76

[7] William A. Woods: Transition network grammars for natural language analysis. Commun. ACM 13(10): 591-606 (1970)