Domain-Independent Reason-Enhanced Controller for Task-ORiented systems - DIRECTOR

Darsana P. Josyula[1], **Michael L. Anderson**[2] and **Don Perlis**[1,2]
(1) Department of Computer Science
(2) Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
{darsana, anderson, perlis}@cs.umd.edu

In today's market, different systems (e.g., camera, microwave etc.) exist that can perform specialized tasks. However, users find it frustrating to learn how to operate these task-oriented systems (TOSs) in a manner that suits their needs. If a single agent can interface users with different TOSs, then the users need not learn how to interact with each TOS separately; instead, they just need to learn how to interact with one agent. In addition, if the agent is rational, users can flexibly adapt the operation of the TOSs by interacting with the agent.

For example, consider a pool controller that can accept commands to heat a pool, stop heating, and provide the temperature. A user interacting directly with such a system will have to issue the appropriate commands when required. By integrating a rational interfacing agent with such a system, the user could tell the agent to heat the pool every Saturday at 10:00 am and maintain the temperature around 35^oC until 1:00 pm. The agent will then issue appropriate commands to the pool controller at the required times.

For such an interfacing agent to effectively control different TOSs it should have the capability not only to translate a user request into a TOS instruction, and to issue that instruction to the TOS at the appropriate time(s), but also to track the effect of those commands and to detect any perturbations, such as contradictory information, or a difference between expected and actual outcomes.

We are developing such a perturbation tolerant, domain-independent interfacing agent (Anderson, Josyula, & Perlis 2003; Josyula, Anderson, & Perlis 2003) by modeling the beliefs, desires, intentions, expectations and achievements of the agent. The current version of the agent has been successfully integrated with and tested on six different TOSs.

Our agent is built on a logical engine (ALFA - Active Logic For Agents) based on Active Logic (Elgot-Drapkin & Perlis 1990; Purang *et al.* 1999; Purang 2001). ALFA can keep track of the evolving time, handle contradictory information and distinguish between desires, intentions and expectations that are achievable and those that are not. Therefore, our agent based on ALFA can rationally deliberate *despite* having deadlines and contradictory beliefs, desires or other mental attitudes. In addition, it can allocate its resources to satisfy achievable goals, resist attempting unachievable goals (until they become achievable) and ignore achieved goals.

Architecture

A general architecture for a rational interfacing agent for task-oriented systems is shown in Fig 1. In DIRECTOR (an acronym for a Domain-Independent Reason-Enhanced Controller for Task-ORiented systems), the KB stores the beliefs (domain-specific as well as general), desires, intentions, expectations and achievements of the agent. Given an utterance, the **parser** asserts a syntactic structure of the utterance in the KB.

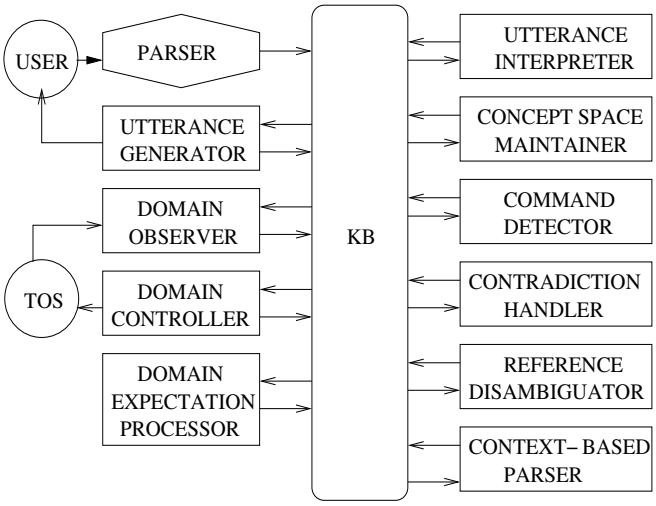

Figure 1: DIRECTOR Architecture

The **Utterance Interpreter** first checks whether there is a valid parse available. If the parse is not valid, then it creates a desire to parse the utterance using the **Context-based Parser**.

Once there is a valid parse available, the **Utterance Interpreter** creates a desire to identify the command in the utterance. The Command Detector identifies the command in a user utterance using the parse and the list of valid commands available in the KB. Once the Detector asserts the command associated with an utterance in the KB, the Interpreter uses the syntax and semantics of the command (from the KB) to

determine the objects associated with the command. In case, the object references in the utterance parse are ambiguous, the interpreter creates a desire to determine the referents for the ambiguous object references.

For each ambiguous object reference, the **Reference Disambiguator** first introspects to determine whether the reference can be disambiguated using the current contents of the KB. If introspection does not resolve the referent, then the Disambiguator creates a desire to get help from either the user or the TOS. For instance, for the ambiguous expression *"the Chicago train"*, the Disambiguator first introspects to determine the referent. If introspection fails to identify the referent, then a desire is created to get help from the TOS. (Thus, the agent could ask the TOS for the current train at Chicago, if it already knows that the expression *"the Chicago train"* refers to the train that is currently at Chicago.) If both introspection and interaction with the TOS fail to provide a referent, then the Disambiguator will create a desire to ask the user for clarification.

Utterance Generator executes the intentions of producing utterances. For questions type of utterances, an expectation about the user's response is created and represented in the KB. This expectation is used by the Context Based Parser to parse user responses.

Domain Controller executes the intentions of sending valid instructions to the TOS. If there is an anticipated result (in the KB) associated with an instruction, then an expectation for that result is created in the KB, when that instruction is issued to the TOS. Once the Domain Controller executes an intention, it creates a desire to inform the user about the action that it is taking.

For each expectation that the Domain Controller produces, the **Domain Expectation Processor** initiates an observation to confirm that expectation, if such an observation action has not been initiated already.

While Domain Controller *sends* instructions to the TOS, its counterpart - **Domain Observer** - *receives* information from the TOS. The information about the format of the TOS outputs and the expectations created by the Domain Controller are used to interpret the observations that the Domain Observer receives.

The **Concept Space Maintainer** makes modifications to the belief space of the KB. These changes include additions, modifications and deletions; correspondingly learning, belief revision and unlearning respectively is achieved. Once the Concept Space Maintainer executes an intention, it creates a desire to inform the user about the action that it has taken.

When a contradiction occurs in the KB, the **Contradiction Handler** notes the contradiction and tries to resolve the contradiction. How the contradiction is resolved depends on the type of contradiction. When enough information is not available to fix an inconsistency, the contradiction handler will create a desire to get help from the user to fix the inconsistency.

Implementation

We have implemented the DIRECTOR architecture in an interfacing agent - ALFRED (Active Logic for Reason-Enhanced Dialog). ALFRED interprets the user requests and creates desires to accomplish the requests. Based on its knowledge and availability of time and resources, it creates and executes intentions to achieve the requests. Since ALFRED does not interact with the real world, the only way it can determine whether an instruction that it issued to a TOS has resulted in the desired action is by confirming it from either the user or the TOS. In order to make this determination, ALFRED creates an expectation regarding the outcome, based on available knowledge, whenever it initiates an action and compares the actual result with the expected outcome.

The current version of ALFRED can translate a user request to a valid action request, for either the TOS to perform a task or for ALFRED to update its Knowledge Base. In addition, it can learn alternative names for existing TOS commands, ALFRED commands or domain objects via user instructions. ALFRED can also interface with different TOSs by merely changing the initial Knowledge Base.

Alfred has been tested on the following domains:

- Simulated Pool: Alfred controls the temperature settings of a pool based on user needs.
- Movie Player: Alfred plays different movies based on user requests.
- Toy Train: Alfred moves various trains to different cities based on user requests.
- Simulated House: Alfred controls different appliances in a house based on user needs.
- Home Designer: Alfred helps the user create and move different objects in a house model.
- Chess Player: Alfred plays chess for the user by sending the user requested moves to a chess program.

References

Anderson, M. L.; Josyula, D.; and Perlis, D. 2003. Talking to Computers. In *Proceedings of the Workshop on Mixed-Initiative Intelligent Systems at the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*.

Elgot-Drapkin, J., and Perlis, D. 1990. Reasoning Situated in Time I: Basic Concepts. *Journal of Experimental and Theoretical Artificial Intelligence* 2(1):75–98.

Josyula, D. P.; Anderson, M. L.; and Perlis, D. 2003. Towards Domain-independent, Task-oriented, Conversational Adequacy. In *Proceedings of the Eighteenth international Joint Conference on Artificial Intelligence (IJCAI-03)*, 1637–1638.

Purang, K.; [Josyula], D. P.; Traum, D.; Andersen, C.; and Perlis, D. 1999. Practical Reasoning and Plan Execution with Active Logic. In *Proceedings of the IJCAI-99 Workshop on Practical Reasoning and Rationality*, 30–38.

Purang, K. 2001. Alma/Carne: Implementation of a Time-situated Meta-reasoner. In *Proceedings of the Thirteenth International Conference on Tools with Artificial Intelligence (ICTAI-01)*, 103–110.