1 Introduction

Traditional methods for Natural Language Processing [1] have failed to deliver the expected performance required in a Conversational Agent. This is mainly due to the fact the speakers rarely use grammatically complete and correct sentences in conversation. Furthermore, speakers make significant assumptions about the background knowledge of the listener. Thus, pragmatic knowledge about the context of the conversation turns out to be a much more important factor in understanding an utterance than traditional linguistic analysis. For this reason, the majority of conversational systems being implemented favour shallow parsing techniques. The challenge for such systems is in managing information about the context of the conversation.

The main premise underlying the use of shallow parsing techniques in conversational agents is that much of human conversation requires little thinking. Many conversations are alike or, at least, have many elements in common. Thus, skill in general chat can be built from a large library of rehearsed responses. Chat agents usually rely on scripts containing sets of rules that consist of patterns and associated responses. Robust conversational agents, i.e. those that are able to deal with a wide variety of inputs, generally have many rules, each of which deals with some variant of an input pattern.

Writing scripts shares many of the same difficulties encountered in knowledge engineering for expert systems. At present, building robust scripts is hard work. New rules are created by hand, possibly with limited automated assistance. When a new rule is added to a large script, it is often difficult to predict how it will interact with existing rules. As in expert systems, it is useful to segregate rules into "contexts" that limit the scope of the rule.

As well as minimising unwanted side effects of new rules, contexts also assist in producing an appropriate response to the user's input. For example. the user inputs a sentence that includes the word "Mars". If the context of the conversation is Solar System then the possible responses will be limited to those about the fourth planet from the sun, rather than about the mythological god. If the context of the conversation is about Alan Turing, then it is most likely that pronouns such as "he" or "his" refer to Turing. Thus anaphoric references can often be handled very easily.

There is a variety of mechanisms for representing and managing context in a conversational agent. In this paper, we describe some of those mechanisms and the approach taken in our system called the "ProBot", which is a scripting engine embedded in a Prolog interpreter. We also describe the history behind the design of the context mechanism and discuss aspects of script writing that have implications for future work.

2 Representing Context

A common approach to dialogue management is to represent the flow of the dialogue as a graph[4]. This is well suited to simple dialogues such as telephone directory assistance where the information being exchanged is well defined. The user is prompted and as each question is answered, the dialogue manager traverses one edge in a graph to take the system to a new state.

Unfortunately, graph-based methods for keeping track of the current context are inflexible and are incapable of dealing with unexpected inputs from the user. To avoid this problem, Thompson and Bliss [8], have developed a frame-based system in which the nodes of the graph are replaced by frames. Since each frame can contain a program, the choice of the nest state can be determined dynamically and is not fixed as in a graph-based system. This is clearly more flexible but is still oriented towards short dialogues in which the range of inputs and outputs is limited, such as the telephone help desk. It is difficult to extend scripts to be able to cope with unexpected inputs on a variety of topics.

Takett and Benson [6] allow the script writer to attach priorities to rules. Some high priority rules will always be checked. These can be used to detect changes in topic or provide default rules, in case no other rule matches the user's input. Rule prioritisation can be taken even further, as in Virtual Personalities Inc's[1] scripting language, in which all scripting rules are assigned an activation level. A side effect of a rule firing is that it can excite or inhibit the firing of other rules, thus implementing a spreading activation model.

Unfortunately, tuning activation levels is a trial-and-error process. As we shall see in the next section, the ProBot attempts to combine the strengths of a flexible graph-based approach with mechanisms for explicitly dealing with unexpected changes of topic and unmatched inputs. The ProBot also differs from previous systems in that it is embedded within a Prolog interpreter, which serves as a deductive database for the conversational agent.

3 The ProBot

The "ProBot" is implemented as a rule-based system embedded in a Prolog interpreter[2]. The rules consist of patterns and responses, where each pattern matches a user's input sentence and the response is an output sentence. Both patterns and responses may have attached Prolog expressions that act as constraints in the patterns and can invoke some action when used in the response.

3.1 Contexts

Rules are grouped into sets, called *contexts*. There is always a "current" context. This is the set of rules that is considered most applicable to the current state of the conversation. A user's input is processed by attempting to match the sentence with the pattern in each rule. The first match causes the corresponding rule's response to be invoked. In addition to outputting an appropriate response, the rule can also invoke a Prolog program. One such program switches the current context. We can represent the course of a conversation as a traversal of a graph in which each node is a context. The rules of the current context are used to carry on the conversation for some period, a transition to a new context corresponds to the traversal of an arc from one node in the graph to another.

[1] http://www.vperson.com

[2] The chat engine is implemented as a component of the author's Prolog interpreter, called, *iProlog.*

While this scheme works well for a highly structured conversation, it fails to take into account sentences that either change the topic of the conversation or those that contain expressions that the rules of the current context are not able to handle. To avoid these problems, we introduce "filters" and "backups". A filter is a set of rules that is prepended to the current context. That is, the filter rules are examined for a match before any of the rules in the current context. Typically, filter rules contain keywords that assist the system in guessing that a user is changing the topic of the conversation. When a filter rule matches, the usual response is to switch the current context. Backup rules are appended to the current context. Their function is to catch user inputs that do not match any rules in the current context. Usually backup rules produce responses that temporise, provoking the user into providing further input that may be recognised by the system.

3.2 Scripting Rules

An example best explains how ProBot rules work. At present, the Museum of Applied Arts and Sciences in Sydney, also called the "Powerhouse Museum", is running a demonstration in which a conversational agent acts as a guide to their "Cyberworlds" exhibition. The demonstration, created by C. Sammut and D. Michie, working on behalf of the Human Computer Learning Foundation (HCLF), covers such topics as the nature of computers and their history, including information on Alan Turing and Charles Babbage (see D. Michie in this volume: "The return of the imitation game"). The present implementation uses the *Infobot* chat engine [2], which was developed by the HCLF and now forms the core of a commercial version *Infochat* now available from Convagent Ltd., Manchester, UK.

In the examples below, we show how a functionally similar set of scripts can be implemented using the ProBot. The only element that the *Infobot* and the Probot have in common is that the Probot uses the Infobot's *response expressions,* described below. The top level scripts are shown in Figure 1.

Scripting rules are of the form `pattern ==> response`. Pattern expressions may contain '*', indicating that zero or more words may match. '~' within a word indicates that zero or more characters may be matched. Alternatives are given by constructs of the form:

```
{alternative1 | alternative2 | ... }
```

Patterns can also contain *non-terminal symbols*, i.e., references to other pattern expressions. This enables the script writer to create abbreviations for common expressions such as lists of alternatives for the various ways in which the user can enter affirmative and negative answers. Since the definitions of non-terminal symbols may be recursive, pattern expressions are equivalent in expressive power to BNF notation.

Response expressions, which are directly derived from the Infobot [2], contain two different types of alternative constructs. Alternatives surrounded by braces ("{", "}") indicate that any element may be chosen at random for output to the user. Alternatives surrounded by brackets ("[", "]") are chosen in sequence. Thus, if the same rule fires more than once, the first alternative is chosen on the first firing, the second element on the second firing, and so on. The sequence repeats when the last alternative is output.

```
museum ::
    #new_topic(museum, museum_topics, eliza)
    init ==>
    [
        Welcome to the Powerhouse Museum and our exhibition
        on the Universal Machine. We can talk about lots of
        things, including Alan Turing and his ideas on
        Artificial Intelligence.
    |
        We have a great exhibit on Charles Babbage and
        computers in general.
    |
        We can talk about other things, like Robotics and
        Machine Learning.
    ]

museum_topics ::
    {* comput~ * | * universal *} ==>
    [
        #goto(universal, [init])
    ]

    {* control * | * information * | * processing *} ==>
    [
        #goto(control, [init])
    ]

    {* communications * | * media *} ==>
    [
        #goto(media, [init])
    ]

    {* AI * | * artificial intelligence *} ==>
    [
        #goto(ai, [init])
    ]

    {* alan * | * turing *} ==>
    [
        #goto(turing, [init])
    ]

    {* charles * | * babbage *} ==>
    [
        #goto(babbage, [init])
    ]

    {* robot~ * | * learn~ *} ==>
    [
        #goto(stumpy, [init])
    ]
```

Figure 1: Top level of scripts that cover some of the topics of the *Infobot* scripts written for the Powerhouse Museum

These constructs may be nested. Thus, response expression can be created so that the user rarely sees the same response twice even when the same rule is fired frequently.

Expressions preceded by a '#', whether they are in the pattern or the response, are passed to Prolog for evaluation. In the pattern, they may be used as conditional expressions to guard firing of the rule. In the response, they may be used to perform some action, such as changing the current context. For example, `#goto(context_name, initial_pattern)` causes the system to switch to the named context and tries to find a rule that matches the given initial pattern.

Rules are grouped into contexts of the form: `context_name :: rule_set`. There is always one context designated as the *current context.* Thus, we may have a particular context that deals with material on Alan Turing (see Figure 2). The current context should contain all the rules necessary for handling a particular state of the conversation. For example, the context, `turing_preamble`, handles the introduction to the material on Alan Turing. Once the user indicates that he or she wishes to known more about Turing, the context is switched to `turing_test`.

We can think of a single context as a node in a graph, where each node represents a particular state of the conversation. As the conversation shifts, the agent can invoke a change of context. A common case where this occurs is when the agent is seeking information from the user. Once a particular piece of information has been obtained, the agent moves on to a new state in which the next piece of information is sought. For example, the agent may wish to learn the user's name, age, etc. We say that the set of contexts that are related form a *topic.* Thus the Turing rules form a topic.

3.3 Filters and Backups

Traversing a graph in the fashion described above is appropriate for a highly structured conversation. But natural conversation often jumps between topics. For example, the user may become bored with Alan Turing and would prefer to find out about the museum's exhibition on robotics. Thus, the agent must have the flexibility to recognise expressions that are out of context and attempt to switch to a more appropriate one. We adopt the convention that when a new topic is entered, the script declares the topic name and also provides the names of two special contexts: the *filter* and the *backup* (Figure 3). The filter contains sets of rules that are always checked before any of the rules in the current context and the backup rules are checked if the none of the rules in the current context match the input. Figure 4 shows a transcript of a session with the museum scripts and illustrates how topics can be changed. In this case, the filter is effectively a jump table that associates topics with particular keywords.

It will often be the case that the script writer cannot anticipate all the possible inputs that a user may give. In such cases, we want the script to recognise a failure to match and prompt the user with some response that will keep him or her engaged and hopefully some further input will get the scripts back on track. The function of the "backup" context is to give a non-committal response that will keep the conversation going.

Filters and backups can be pushed onto a stack. In our museum example, at the start, we push onto the stack of backup contexts a set of "Eliza" rules.

```
turing ::
        #new_topic(turing, [], turing_backup)
        init ==> [Alan Turing is my hero!]

        { * why * | * really * | * weird * | * strange *}==>
                #goto(turing_preamble, [init])

turing_preamble ::
        init ==>
        [
                As early as the mid-1930's Alan Turing had the underlying
                mathematical idea on which the computers of today's are
                based.  He was also one of the founders of what we now call
                Artificial Intelligence or AI. he invented a test for
                intelligence.  Shall I describe it?
        ]

        <aff> ==> [#goto(turing_test, [init])]

        <neg> ==>
        [
                Well, I think Alan Turing was an interesting person,
                but we can move on.
                #pop_topic([init])
        ]


turing_test ::
        init ==>
        [
                Turing was the first to put up a testable
                definition of Artificial Intelligence.  He
                phrased it as an "imitation game", and it
                has come to be known as the Turing Test.
        ]

        {
          * describe * | * what is it * | * what * test | * tell *
        } ==>
        [
                Turing's "imitation game" was designed for detecting
                the presence of intelligent thought processes in computer
                systems.  It has become known as "The Turing Test".
                The candidate program is interrogated alongsidea human.
                Both are remotely connected to a human examiner.  If the
                latter can't score better than seventy per cent correct
                in spotting which is the computer and which is the human,
                then the machine has demonstrated its possession of some
                some degree of intelligence.  The same paper, which
                appeared in 1950, contained an extraordinary proposal.
                This "child machine" project was overlooked, and remains
                so to this day.
        ]

        { * proposal * | * what * it * | * what * that * | * what * was *
        | * what * did * | * child * machine * | * what * project *
        } ==>
        [
                The proposal considers how precisely are we to design and
                build a machine with human-type and human-level
                intelligence.  His recipe was first, to develop a system
                that can learnfrom experience, from instruction and from
                imitation.  Thenspend as long as it takes to train it and
                educate it to human standards.
        ]

turing_backup ::
        * something {else | different} * ==> #pop_topic([init])
```

Figure 2: ProBot scripts that contain material on Alan Turing.

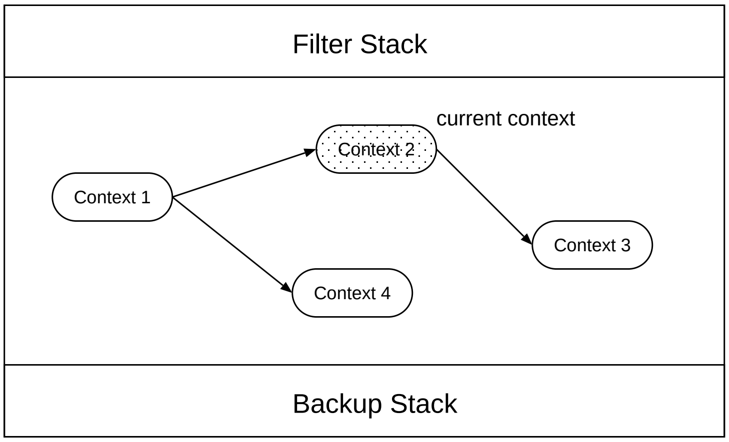

Figure 3: Topic structure

Welcome to the Powerhouse Museum and our exhibition on the Universal Machine. We can talk about lot's of things, including Alan Turing and his ideas on Artificial Intelligence.

Who is Alan Turing?

Alan Turing is my hero!

Why?

As early as the mid-1930's Alan Turing had the underlying mathematical idea on which the computers of today's are based. He was also one of the founders of what we know call Artificial Intelligence or AI. He invented a test for intelligence. Shall I describe it?

Yes, please.

Turing was the first to put up a testable definition of Artifcial Intelligence. He phrased it as an "imitation game", and it has come to be known as the Turing Test.

Let's talk about robots, instead.

Have you seen Stumpy, the six-legged robot, learning to walk?

no.

You should go see it. It's fun. It's a insect-like robot that learns how to walk. Do you want to know how it learns?

sure.

Figure 4: Transcript of session with ProBot scripts. User inputs are in italics.

```
My name is * ==>
[
    Nice to meet you, ^1.    #(add ^1 to name of conversant)
]

What is my name ==>
[
    Your name is ^(name of conversant).
]
```

Figure 5: Using Prolog to store simple facts.

This is a close copy of Weizenbaum's original chat program [7]. It is used here to illustrate a "last resort". When all else fails, string the user along with questions. On entry to the Turing topic another context is pushed onto the backup stack. This is intended to handle user inputs like, "Let's talk about something else". The response associated with this rule includes an action to pop the filter and backup stacks and to return to the previous topic.

3.4 The Database

The chat engine, as described so far, provides facilities for writing quite complex scripts, but all the knowledge of the agent is contained entirely within the scripting rules. Since the ProBot is embedded within a Prolog interpreter, it is easy to extend the capabilities of the chat agent by using Prolog as a deductive database for storing information about the user and about the conversation. A few simple examples illustrate this facility (Figure 5).

When a wild card ('*') or non-terminal expression matches some user input, the matching sequence of words is mapped to a numbered variable, $\hat{}n$, which corresponds to the n^{th} occurrence of a '*' or non-terminal expression. Numbered variables can be used in the response, either to emit a sentence containing words from the user's input or their values can be used in a Prolog expression. The first rule in Figure 5 uses the numbered variable for both purposes.

The Prolog expression "add ^1 to name of conversant", stores the name of the user in a frame system implemented in *iProlog*. The second rule shows how information can be retrieved from a frame and inserted into the output stream. In general, any Prolog program can be invoked by a response expression. If it is preceded by a '#' then the Prolog program works by side effect, such as asserting something into Prolog's database. If it is preceded by a '^' then the expression is evaluated and the result is interpolated into the output string.

When Prolog expressions appear in the pattern of a rule, they are evaluated as predicates. If the Prolog predicate fails then the pattern match also fails.

4 Managing Contexts

Our original approach to handling contexts in the *Infobot* [2] used a spreading activation model . In this model, each rule is assigned a *base activation level*, a number between 0 and 1. After a rule fires, the activation level is decreased and over time, it gradually returns to its original level. The activation level is used to resolve conflicts when two or more rules have patterns that match the user's input. Thus, a rule with a higher activation level are more likely to be chosen than a rule with a lower activation level. When a rule fires, it may "promote" or "demote" other rules by raising or lowering their activation levels. So in the museum example, if we find that the user is interested in Alan Turing, we would promote all the rules associated with Turing.

The use of activation levels and the numerical ranking of rule priorities places a large burden on the script writer. As the number of rules in the script grows, it becomes difficult to remember the rankings, so the addition of new rules becomes problematic as unexpected interactions between rules can occur. There are several ways of trying to minimise this problem. An intelligent editor may assist by keeping track of rule priorities and presenting them to the script writer. The system may also maintain a set of test cases for verifying the new rule [6]. The latter method, from NativeMinds Inc, does not use activation levels, but "priority" and "default" rules similar to the filters and backups described here.

In the activation level model it is more common to promote and demote sets of rules, rather than individual rules. So in addition to avoiding the problems of ranking rules, we also wish to have a convenient way of invoking sets of rules. For these reasons, we found it useful, in the ProBot, to separate the rules into the small groups that we call contexts. Explicit transitions between contexts allows the script writer to create goal-directed behaviour in the conversational agent, while the addition of filters and backups gives the agent flexibility in handling unexpected inputs or changes of topic. Filters and backups are crucial since, without them, the user could become trapped within a topic and have no way of escaping unless a rule in that topic makes an explicit change of context. A related method is described by Thompson and Bliss [8], in which they manage spoken dialogues by organising rules in a frame structure.

In the museum example of Figure 1, we have adopted a particular discipline of a "dispatcher". That is, we have a jump table that matches keywords with topics. This is used as a filter, but is also useful when a topic has run its course and the agent must look for new things to talk about. In this case, we "pop" the current topic off the stack and return to the dispatcher, which prompts the user for some new input that will give the agent a clue about where to go next.

Note that in Figure 1, the "init" rule has a sequential response expression. The first time the rule is invoked, on entry into the top level context, the system issues a greeting. The next time we re-enter the topic and invoke the "init" rule, the second element of the sequence is chosen and the user receives a prompt to try a new topic. This process is repeated for all the elements in the sequence.

5 Evaluation

Evaluating a conversational agent is not a simple task. In other areas, such as Machine Learning, libraries of test data can be used to put the program through its paces. However, conversational agents are interactive systems, thus, the only method for evaluating their performance is through extensive trials with human subjects. Needless to say, this is time consuming and difficult. At present, no such tests have been done using the ProBot. However, some experiments, by Hayes-Michie [3], have been carried out with the *Infobot*. We briefly describe these experiments to give an indication of the methodologies that can be used. We do not suggest the that results obtained for the *Infobot* in any way give a measure of performance of the ProBot. However, it is intended that similar experiments will be run with the ProBot. We gratefully acknowledge the contribution of Jean Hayes-Michie in what follows.

In Hayes-Michie's pilot experiments, seven undergraduate psychology students (four male and three female) were each asked to hold two seven-minute conversations with the Infobot Museum Scripts. Transcripts of the conversations were recorded and given to the subject for comment. The subject was also asked to rate the amount of time in the conversations when they could imagine that they were talking to a real person. They also rated various aspects of the agent's personality, on a five-point scale, for each of eight qualities: aggressive, friendly, cheerful, amusing, intelligent, human, confident, interesting. Averaging the results, the scores indicated that the subjects found the Museum Guide to be reasonably intelligent and friendly, but not very amusing and only moderately human-like. The agent was able to chat generally, but only to a limited degree and with lapses in which it would lose the thread of the conversation.

Note that an evaluation such as this is an indication of the quality of the scripts. The quality of the scripting language and the underlying engine can only be measured indirectly by the ease with which good scripts can be constructed. To do so would require some measure of the script writer's productivity.

Another tool that is currently being used to evaluate the InfoBot is that we record transcripts of all the conversations that are being held in the Powerhouse Museum itself. We are able to obtain samples of these conversations and examine them for mistakes. The result of this process is usually to expand the patterns and responses in the scripts. We have only recently begun logging the sessions at the Museum so this is an ongoing process. As mentioned previously, we are currently only evaluating the InfoBot, which is a more mature program. The ProBot will eventually be subjected to similar procedures.

6 Discussion

Writing scripts is hard work. Even with intelligent editing and testing tools, the script writer must try to anticipate the large variety of ways in which it is possible to say the same thing. The script writer must also try to anticipate the many ways in which conversations can diverge and find techniques for either following the diversion or of bring the conversation back on track in a subtle and friendly manner.

It is also a simple fact that, in many environments, conversational agents must deal with hostile users. For example, the original scripts for the museum guide assumed a cooperative user who wished to learn more about the exhibits. However, after installation, it was found that the majority of the users were young children who amused themselves by finding ways of making the guide say odd things. A revision of the scripts attempts to make the scripts more robust. As this paper is being written, logs of conversations with the museum guide are being recorded so that the scripts can be further improved.

It is obviously desirable to have a conversational agent that resembles Turing's Child Machine [5]. That is, a program that is teachable and learns from examples. The difficulty at the present stage of development of conversational agents is deciding what can, in practice, be learned. A fully teachable system is, at present, out of the question. However, it may be possible to isolate sub-problems, such as managing the priorities of rules, generalising or specialising the patterns of the rules, or enriching the responses of rules. The task of learning is further complicated by the fact that script writing is not simply about constructing pattern matching rules, but most crucially, it is also about understanding the context of the conversation.

In the present state of our project, we are still learning (for ourselves) what are useful mechanisms for constructing conversational agents. In the course of this investigation, we are also learning what will be useful for the system itself to learn.

Acknowledgement

This work is the result of an ongoing collaboration with Donald Michie and Jean Hayes-Michie. The design of the ProBot has benefitted greatly from our joint experience in the design of HCLF's (now Convagent Ltd's) *Infobot* and from the many discussions that we have had about scripting style.

References

[1] J. Allen. *Natural Language Understanding.* The Benjamin/Cummings Publishing Company, Inc, 1995.

[2] D. Michie and C. Sammut. Infochat Scripter's Manual. Technical report, Convagent, Ltd, Manchester, UK, 2001.

[3] J. Hayes-Michie. Report on Pilot Experiment. Technical report, University of New South Wales, 10 May 2001.

[4] A. Rudnicky and W. Xu. An agenda-based dialog management architecture for spoken language systems. In *Proceedings of ASRU*, 1999.

[5] A.M. Turing. Computing Machinery and Intelligence. *Mind*, 59(236):433–460, October 1950.

[6] Walter A. Tackett and Scott S. Benson. System and method for automatically verifying the performance of a virtual robot. United States Patent No. 6,259,969, 2001.

[7] J. Weizenbaum. ELIZA - A Computer Program For the Study of Natural Language Communication Between Man and Machine. *Communications of the ACM*, 9(1):36–35, January 1966.

[8] William Thompson and Harry Bliss. A Declarative Framework for Building Computational Dialog Modules. In *International Conference on Spoken Language Processing*, 2000.