TOWARD CONVERSATIONAL CAPACITY IN SYNTHETIC CHARACTERS

By

Stephen Nerbetski

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE
MAJOR SUBJECT: COMPUTER SCIENCE

Approved:

_____

Selmer Bringsjord
Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

November 2007
(For Graduation December 2007)

CONTENTS

LIST OF TABLES

ABSTRACT

This thesis expands on two major parts of Natural Language Generation (NLG) in relation to advanced synthetic characters. After a brief overview of previous work, new methods and ideas are explored through the situation of a synthetic character known as E. This document first looks at the process of a synthetic character selecting what it wants to say, and how 'Response Classifications' can help generalize this process. Then, we go on to introduce a new translation engine, based off of previous work, which uses a simple, yet expressive model to translate from our character's knowledgebase representation to English.

1. INTRODUCTION TO SYNTHETIC CHARACTERS

1.1 What is a Synthetic Character?

Put quite simply, a synthetic character is a character which is entirely digital. Generally, the character acts on its own, responding to certain events in the virtual environment without any direct control from a human. A common example of a synthetic character would be many non-player characters seen in video games.

But, there is a limitation with the current approach. Generally, synthetic characters that we see today are only designed to be believable within a certain context; usually, this context is the specific environments the character is originally designed to exist within, such as the official story campaign in most video games. Sometimes, these characters act strangely even within such an environment, due to a circumstance or an action by another agent that the developers didn't anticipate.

The answer to such issues is to generalize the approach; this is where 'advanced' synthetic characters come in. The idea behind an advanced synthetic character is, rather than scripting the character to respond to certain events in certain ways, the character is given a way of reasoning on its own to decide how it should act. Additionally, in order to further the appearance of a character being more than just a computer-controlled avatar, an advanced synthetic character will benefit greatly from being able to communicate on a human level. This is the approach I plan to apply to NLG for advanced synthetic characters in this thesis.

1.2 Previous Work with Advanced Synthetic Characters

In this section I will give a brief overview of some of the previous and ongoing general work with synthetic characters.

1.2.1 MIT — Rea, Etc.

Cassel et al., from the MIT Media Laboratory, argue in a 1999 paper for the importance of embodiment for a synthetic character. In this paper, an agent known as Rea is introduced. Emphasis is put on how important Rea's physical appearance

and mannerisms are toward giving the presentation of a convincing character, as well as the importance of Rea's ability to read a person's mannerisms and interpret what they mean. Rea is further designed to conduct a "mixed initiative conversation" with another agent, although in a limited context. The main focus for this was looking at how the flow of conversation is handled, and how it can be dealt with in a synthetic character.[6]

Additionally, there was the Synthetic Characters Group at MIT. This group dealt with creating AI characters, aiming for the intelligence level around a pet animal. However, the group has been inactive since 2004. One of the latest projects from this group was an autonomous, animated dog which could be trained using a process called "clicker training," resulting in a virtual agent that could learn to respond to sounds with certain actions.[8]

1.2.2 USC — Virtual Humans

The Virtual Humans research group at University of Southern California "... specializes in behavior and emotional modeling, developing human-like software agents for virtual training environments." The group works with multiple areas, including emotion, natural language, and embodiment.[12]

As an example, in researching emotions for virtual humans, the group has the goal of using "... VH agents to investigate how people interpret emotional behavior and how these interpretations influence memory and decision making." They plan to use this to help create compelling agents in virtual environments; they specifically mention virtual training environments as a particularly important target.

1.2.3 Novamente — AGI, NCE, and NIVA

Novamente is a company pushing research into what they refer to as 'Artificial General Intelligence'; this concept refers to the idea of pulling together the typically fragmented fields that are covered by the term 'artificial intelligence', to work toward a " ... whole-systems-focused ..." approach to creating generic artificially intelligent entities. One of the major projects that is being researched is the Novamente Cognition Engine.[3]

The Novamente Cognition Engine, or NCE for short, is "... a software system designed for large-scale implementation on a distributed network ... founded on a unique AGI design grounded in a systems theory of intelligence." The overall goal is to create an agent with general intelligence, to the extent of and even surpassing, human level. At the current point in time, the NCE is being used with a humanoid agent in a virtual environment. This agent is being taught simple behaviors, including recognizing objects by name and playing fetch.[4]

One system the NCE is used in is NIVA, the Novamente Intelligent Virtual Agents Server. NIVA is aimed at creating believable synthetic characters in virtual worlds, using the NCE to display agents that act according to certain rules and toward certain goals, but can be flexible in such restriction based on the situation. NIVA agents are able to learn and reason about situations, and use these in combination with the given goals and rules to determine how to act. NIVA is currently planned for launch, to be used in various applications such as massively multiplayer online games, in early 2008.[9]

1.3 Communicating with Advanced Synthetic Characters

In order to enable an advanced synthetic character to communicate at a human level, there are certain components that must be developed. A typical synthetic character, in the process of communicating, goes through a process somewhat like what follows:

First, some stimulus, such as another entity (possibly a human) communicating with the character or the character simply observing something in its virtual environment, triggers the character into communication. From there, the character goes into some sort of algorithm which determines how the character should respond; this results in the character communicating. For usual synthetic characters, these pieces are hard-coded to specific sets of situations and responses to those situations.

For an advanced synthetic character, however, we need to move away from this pre-designed approach, toward a more dynamic one. For the case of a synthetic character communicating at the human level, this process can roughly be divided into two parts. The first part is Natural Language Understanding (NLU), which

deals with taking an input human-language and parsing it into something that the system can understand. The second part is NLG, which deals, conversely, with taking the input as the system represents it internally, generating a response, and changing it into a human-language format. This part is the main focus of the discussion in this thesis.

2. NATURAL LANGUAGE GENERATION

As mentioned in the previous chapter, NLG is the process of a system generating some output, with the result being in a human-language formulation.

2.1 Previous work in Natural Language Generation

One of the major researchers in the field of NLG is Eduard Hovy. Hovy has authored and co-authored numerous papers on the subject.

One major example is found in the University of Southern California Encyclopedia of Computer Science, specifically the entry on Language Generation. A major part of this comes from how Hovy divided the process of NLG into two major sections. The first section, called Macroplanning, deals with deciding what to say. The second section, called Microplanning and Realization, deals with deciding how to say it. This distinction, as you will see later, is a primary hinge on which this thesis's research rests.[7]

2.1.1 USC — Virtual Humans

As part of USC's Virtual Human project, the group is researching Natural Language Processing (NLP), which includes both understanding and generation. The goals they are working toward are mainly for integration into their virtual training environment project. One specific point they emphasize is that they are simultaneously working on both the input and output ends, a fact they claim is a better approach than focusing on one or the other solely.

2.1.2 Athena.NET

Athena.NET is an interactive development environment for the type-ω denotational proof language (DPL), Athena. Athena was first developed Konstantine Arkoudas as an example of a DPL. A denotational proof language is designed for use in expressing and checking proofs. In other words, these languages allow a user to input a proof, and have the machine run through it and verify that it is valid. A

type-ω DPL, such as Athena, also allows proof searching and computation, meaning that it can be instructed to find a proof of a given statement, as well as perform other more complex computations. A type-ω DPL is effectively a full, Turing-complete programming language with a proof framework.[1]

One useful tool in Athena is methods. A method is a way to write a program in Athena, giving it directions for performing a certain task, such as finding a proof using a specific process.

Athena.NET, developed in 2006 by Sangeet Khemlani, provides a simple development environment for using Athena. However, there is one additional feature in Athena.NET of importance: a translation mechanism. A user can associate various Athena atoms with English phrases, and then Athena.NET, from a proof structure, can generate an English text.[2]

To demonstrate this, here is the built-in example that Athena.NET gives:

```
(declare newyork Boolean)
(declare chicago Boolean)
(declare millions Boolean)
(declare Die (-> (Boolean) Boolean))
(declare Target (-> (Boolean) Boolean))

(assume (or (Target chicago) (Target newyork))
  (assume (if (Target chicago) (Die millions))
    (assume (if (Target newyork) (Die millions))
      (!cd (or (Target chicago) (Target newyork))
           (if (Target chicago) (Die millions))
           (if (Target newyork) (Die millions))))))
```

This is a fairly straightforward proof. It works on a simple scenario, where either New York or Chicago is the target of some attack, and gives a result that, whichever is targetted, millions will die. When put through Athena.NET's translation engine, the following English is returned:

```
Assume Chicago is a target \/ New York is a target.
  Also assume if Chicago is a target then millions will die.
    Also assume if New York is a target then millions will die.
      Recall that Chicago is a target \/ New York is a target. Each case
      produces the same conclusion; that is, if Chicago is a target then
      millions will die and if New York is a target then millions will die. We
      can conclude that millions will die.
```

Aside from the word 'or' being written as the logical symbol \/, this is a fairly accurate transcription of the above proof to english.

This approach is simple and straightforward, although somewhat limited in scope. First, Athena.NET only allows a single English phrase for each atom in Athena, resulting in limited variety. Additionally, there is no mechanism by which an atom can be converted to different phrases based on how it is used in a sentence, which makes some variations result in very odd-sounding phrases as a result. Finally, Athena.NET limits the translation to only Athena constructions which are based on a proof, meaning a user cannot translate an arbitrary atom or construction, but only a limited set of constructions. In the above example, the proof constructions used are `assume` and `!cd`.

2.2 An Approach to Natural Language Generation

The approach taken to NLG in this thesis is as follows. We have an agent, a synthetic character whose 'intelligence core' is a logic engine, which is set in a specific situation to give responses. This agent is discussed in detail in the next chapter.

In this thesis, the methodology used in Athena.NET's translation mechanism is expanded upon, to create a more general translation engine aimed for use with an advanced synthetic character. While it is not going to push completely to the goal of fully functioning NLG, the goal of this thesis is to explore further into this approach and work toward a more complete NLG methodology for synthetic characters. This is achieved by looking at generalizing the current approach further, leading toward a more generally applicable approach to converting logic output to English.

Additionally, this thesis looks at the first part of NLG, that of an agent deciding what to say. Again, this thesis will not fully solve this part of the overall problem, but instead will push toward a more general approach that can be applied for advanced synthetic characters.

3. E — A SYNTHETIC CHARACTER

E is a synthetic character, used to demonstrate concepts of advanced synthetic characters.

E was previously discussed by Bringsjord et al. in 2005. In that paper, E is introduced as a character which is built from the definition of being 'evil'. The definition presented in that paper is as follows:

> Person s is evil iff there exists some action or omission such that
>
> 1. performing a is morally wrong;
> 2. s is morally blameworthy for performing a;
> 3. s performs a in the hopes of causing considerable harm to others; and
> 4. were s a willing and open participant in the analysis of reasons and motives for s's seeking to perform a, it would be revealed that either
>
>   **(i)** these reasons and motives are unintelligible, or
>
>   **(ii)** s seeks to perform a in the service of goal g, and
>
>   **(a)** the anticipatable side-effects e of performing a are bad, but s cannot grasp this, or
>
>   **(b)** g itself is appraised as good by s when it is in fact bad.

One important part of E is the situation he is given. This situation is based on one presented in the book *People of the Lie* by M. Scott Peck. E takes on the role of a father, who, after his older son shoots himself, gives the same gun to his younger son, Bobby, at Christmas, reasoning that it would make Bobby happy since he loves to hunt. However, Bobby instead spirals into decline, while E is left puzzled as to why Bobby wouldn't appreciate the gun. The interaction with E comes at a later point, when E sits down to talk about Bobby's situation with the user (who takes the role of a psychologist). Through a dialogue, it is found that, while E had thought giving Bobby the gun would make him happy, E could simultaneously rationalize it making Bobby unhappy; such fits the definition of evil proposed in the

paper, especially the clause that "... (i) these reasons and motives are unintelligible ..."[5, 10]

There was originally a demonstration of this agent, at the GameOn conference in 2005. However, the implementation of this demo lacks much in the way of generality and flexibility; this is one of the main points upon which the present work improves.

4. IMPROVING E

This thesis looks at both parts of NLG. First, there is a look at text planning, and then there is discussion on text generation.

4.1 Text Planning — Response Classifications

The first part of NLG is planning what is to be said. In looking at E's scenario, I used what I refer to as 'Response Classifications'. A response classification is a general way of responding to a certain class of input stimuli; in the case of E, these stimuli are all various things said by the user. For this system to work, the input system, which includes NLU, would need to determine what response class is appropriate for a given input.

In developing E, I determined five response classifications, listed in the table below. Note that this list is not exhaustive; these are only the ones that were found while working with the dialog scenario with E.

Table 4.1: A List of Response Classifications

Response Classification	Description
Claim Check	A simple check of validity, based on the agent's KB
Why	A request for the reasoning behind a certain statement
Fill-in-the-Blank	Trying to find a certain item that fits a description
Permission	Requesting the agent's permission for something
Conversation Rule	A simple structure for conversation.

These are discussed in the following sections.

4.1.1 Claim Check

The 'claim check' response class is probably the simplest response class. It is simply asking the agent whether or not something is true. For example, an agent would use a claim check to respond to the question, "Is it cloudy in Troy today?"

In the KB for E, the claim check is implemented using a simple call to the theorem prover. If the KB can show that a statement is true, it returns that statement, stating it as a theorem. The only issue that comes up with this implementation is if the statement cannot be shown to be true; on failure to prove something, Athena only returns that it could not prove the statement. This limits what an agent can say in response; either we will need to use a generic response, or we will need to see if we can build a countermodel, by asking Athena to prove that the intended statement is not true instead.

The claim check response classification is implemented in E's knowledgebase as the method E_ClaimCheck.

4.1.2 Why

A 'why' response class is pretty simple: the agent is being asked to generate reasoning for something. As an example, one could ask the character a question such as, "Why did you give him the gun?"

Unfortunately, implementation of this type of response class is not in the current KB. This is because Athena does not have a mechanism for returning the reasoning behind a result, which I see as a fundamental necessity for this type of response. To implement the 'why' classification, one would need to use a theorem prover that not only returns the result, but gives a rationalization for these results.

To illustrate how this could work, an arbitrary method called 'show_why' will be used. This is not an actual method in Athena, but it is used here for demonstration purposes. It will output not only that a given statement can be proven, but also the logic behind why it is provable. Assume, for instance, an agent knows the following to be true:

```
(has p_Bobby o_gun)       # Bobby has a gun.
(if (has p_Bobby o_gun)   # If Bobby has a gun
    (is p_Bobby a_happy)) # then Bobby is happy.
```

We then want to know why the agent thinks Bobby is happy:

```
Input: (!show_why (is p_Bobby a_happy))
Output:
1> (has p_Bobby o_gun)         [KB]
```

```
2> (if (has p_Bobby o_gun)
       (is p_Bobby a_happy))   [KB]
3> (is p_Bobby a_happy)        [MP, 1, 2]
```

The output here is in an arbitrary formatting for example purposes. The first two lines are pulled directly from the assumption base. The third shows how the conclusion is drawn: The theorem prover uses modus ponens with statements 1 and 2 to achieve the result. This full proof form could be used to allow an advanced synthetic character to generate an explanation as to why something is true.

4.1.3 Fill-in-the-Blank

A 'fill-in-the-blank' response classification is pretty much what it sounds like — the agent is filling in a piece of information given other information around it. One example of this would be asking the agent, "Who is in the room with you?"

To implement this type of response class, the agent needs, quite simply, to fill in an existential quantifier. In the latest version of Athena, this can be handled using the '!find_model' command; unfortunately, I could not get the command line version of Athena working, and could only rely on Athena.NET, which uses an older version of Athena, and therefore does not include this command.

As an example of how this could work, assume an agent knows that Bobby's brother used to have the gun:

```
(used_to_have p_Bobbys_brother o_gun)
```

We could then ask the agent "Who used to have the gun?"

```
Input: (!find_model (exists ?a (used_to_have ?a o_gun)))
Output:
?a = p_Bobbys_brother
(used_to_have p_Bobbys_brother o_gun))
```

The agent finds that it can satisfy the existential quantifier, and returns that Bobby's brother used to have the gun.

Another way to implement this is to use another automated theorem prover, known as SNARK. We can simply ask SNARK to prove that there is some entity which had the gun:

```
Input: (prove '(used_to_have ?a o_gun))
Output:
(|SNARK-USER|::|USED_TO_HAVE| |SNARK-USER|::|P_BOBBYS_BROTHER|
 |SNARK-USER|::|O_GUN|)
   |SNARK|::|ASSERTION|)
```

In the input, the `?a` is a symbol representing an unknown entity which the user wants SNARK to find. The output shows us that SNARK did find that Bobby's older brother used to have the gun.[11]

4.1.4 Permission

A 'permission' response is basically what it sounds like. The agent is being asked if it cares if something happens or is done. For instance, "Do you mind if I turn on the radio?" This response class could also be triggered in nonverbal ways; for example, if the agent sees a person nearby pull out a cigarette, the agent could run a permission check, possibly resulting in the agent asking the person not to smoke nearby.

The current implementation in E's knowledgebase is actually a stub. It is designed to fit the realm of the conversation and nothing more. The implementation of this is in E_Permission.

To generalize this, one would need to implement a system where the agent, upon receiving input triggering a permission response class, will need to run a query on the theorem prover to check a statement such as, "If X occurs, will there be some negative consequence for me?" If this returns true, then the agent will need to formulate some way of expressing disapproval for the action in question; otherwise, the agent should respond, giving permission, if the agent was asked explicitly for permission.

4.1.5 Conversation Rule

These response classifications are by no means only for when the agent is asked a question. The 'conversation rule' classification is one such example. This represents certain responses that are simply expected by the usual flow of conversation; for example, when someone says "Hello", generally you will greet them in return.

Because of the nature of this response class, it is the exception in that most of its cases will need to be checked explicitly. Fortunately, these cases tend to be simple, and there are not too many of them. In E's KB, the only case implemented is the case for "Thank you." In return, E gives a symbol representing "You're welcome." This is implemented in the method E_ConvRule.

Other conversation rules that will be needed in a more robust character include rules for "Hello" and "Goodbye". Additionally, one could augment some of these rules to allow more specific detail, such as dealing with "Thank you for letting me borrow your umbrella."

4.2 Text Generation — A Translation Engine

Once the agent determines what it wants to say, it then needs to actually express it. Generating the text is done by feeding the result of the theorem prover to a translation engine.

Based on the model used in the Athena.NET English translation function, I created a Java program that translates the output from Athena into English sentences. The design I implemented has some advantages over Athena.NET's implementation.

First, my design allows the translation of any arbitrary symbol. This allows a much greater freedom of expression than Athena.NET's design, which requires that any translated text must be within some proof structure.

Second, my design incorporates the ability to allow multiple phrases for a single term; Athena.NET only allows a single English phrase for each term. There are two ways this is achieved: by allowing multiple phrases to be explicitly defined, and by giving the ability to use pronouns. This allows for a much wider range of expressivity, and gives the synthetic character a more human feel.

Additionally, as part of implementing the above, the new translation engine also supports multiple parts of speech, leading to a more natural result. For example, you cannot, without changing the definitions in between, use Athena.NET to translate `(gives_to p_Bobby o_gun p_Me)` to "Bobby gives the gun to me." and `(gives_to p_Me o_gun p_Bobby)` to "I give the gun to Bobby." With the new trans-

lation engine, it can understand the difference between “I”, “me”, “my”, “myself”, and such, and use them appropriately.

Note: This section describes the methodology behind the translation engine. For instructions on using the translation engine itself, please see Appendix D.

4.2.1 Inner Workings

The translation engine I designed works by separating the given input into terms, and translating each term. In the translation engine, a term is a single unit which has a certain meaning, expressed in the representation output by Athena. A term can be either a simple term or a complex term.

A simple term is a single atom; examples of this include p_Bobby, a_ingrateful, o_gun, and so on. When translating a simple term, the translator uses its lexicon of known terms to look up a meaning for the term. It may also choose to use a pronoun for the term instead. The resulting text is returned as a string.

An example of a simple term:

```
p_Bobby
Bobby.
```

When the user enters the atom p_Bobby, the translator finds the term and chooses an English phrase to represent it. The result is displayed.

A complex term, on the other hand, leads to more interesting and expressive forms. A complex term uses parentheses to indicate ‘function’ atoms. The following is an example of a complex term using E’s knowledgebase. Note that ‘t2’ is an atom representing the point in time of Christmas.

```
(gives_to p_Bobby o_car p_U t2)
Bobby gave a car to you at Christmas.
```

The first symbol in a complex term is the important one; it acts as a function while the rest of the symbols act as arguments. In this case, the function is gives_to, with the arguments p_Bobby, o_car, p_U, and t2. When processing this, the translator first chooses a phrase for the function term, based on its lexicon. Then, it recursively translates each argument, and uses them to fill in pieces of the selected phrase. Note that complex terms can have complex terms within them as

Table 4.2: Data Fields for each Term

Field	Description
term	The logic language atom represented by this term
phrase	One or more natural language phrases that represent this term
gender	Assigns a gender to this term
pronoun class	Determines the type of pronouns to use

arguments, as shown in this example from another KB:

```
(if (target p_newyork) (die millions))
If New York is a target, then millions will die.
```

In this example, we have nested two function terms, 'target' and 'die', inside another one, 'if'. It should be noted that, while 'if' is technically a logical construction, for the purposes of the translation engine it is interpreted as a term.

4.2.2 The Lexicon

In order for the translation engine to work, it needs a lexicon of terms and their associated English phrases, as well as some extra information to ensure proper usage. The lexicon is the translator's data structure to store this information. With the current implementation, the lexicon is statically defined in one or more text files, and loaded when the translator is initialized.

The lexicon consists of a list of terms, each with multiple pieces of data:

When a term is selected for translation, the translator will either choose to use a pronoun for the term, or select one of the phrases for the term, at random.

For the actual format of the lexicon file, please see Appendix A.

4.2.3 Pronouns

The translator has a very important tool for using pronouns, namely the 'pronoun class'. This determines which pronouns, if any, the translator should use for the term.

For each term, the lexicon defaults to not using pronouns for a term. To have the translator use a term, you need to give it a pronoun class within the lexicon file.

Table 4.3: Pronoun Classes

Class Number	Description
0	Do not use pronouns
1	First-person (I, me, ...)
2	Second-person (You, your, ...)
3	Third-person (He, she, it, ...)

The current rules for using pronouns are a bit naive; first- and second-person pronouns are always preferred over using the actual phrases, while third person pronouns are used if the actual phrase was used within the past 2 to 5 references to the term. The rules for first- and second-person pronouns are not far off by typical standards for English speaking; the only major exception being that a person's name is often used instead of the word 'you' for calling out to someone. The third person pronoun usage rule is an oversimplification. It is based off of the idea that pronouns are used when they can be understood as referring to a specific object; however, this set only relies on simply the number of mentions since the last time the actual phrase was used, while in reality there are more complex underpinnings to this rule.

4.2.4 Parts of Speech

While designing the mechanism for pronouns, it became apparent that, in order for a virtual agent to use pronouns properly, they must have a grasp on the concept of parts of speech. In the implementation presented in this thesis, there are four parts of speech considered: declarative, self-reflexive, object, and possessive.

The declarative part of speech is the simplest. It is basically just naming something. For example, "Bobby" is a declarative form.

The self-reflexive part of speech is when something refers back to itself; this is usually done with a pronoun. "Myself" is a simple example of this.

The 'object' part of speech actually encompasses both the direct and indirect object parts of speech. Since these two result in the same word form, they are grouped together in this program. In the sentence, "I gave the it to him," both "it" and "him" are object forms.

The possessive form is that of having something, indicating that one noun belongs to another. In the current implementation, this is the only form that will modify a given phrase from the lexicon; all others only differ in pronouns. As an example, in the phrase, "Bobby's gun," "Bobby's" is the possessive form of "Bobby".

To use these parts of speech, a bit of additional information is given to the lexicon for complex terms. This extra information indicates which part of speech a recursively processed argument term is to be treated as. There is also the ability to explicitly specify that a pronoun should always be used; this is useful for a phrase such as "Bobby's brother killed himself." The "himself" part is best left always as a pronoun, since the subject is always stated right before in the same phrase.

4.3 Results — The New Conversation with E

With the new translation engine and the updated knowledge base, here we run through the script for the dialogue with E. Please note that each line follows the format:

```
U: Something the user says
Runs: The call to Athena
Returns: The theorem returned from Athena
E: What E says, based on the output
```

'U' is a generic alias for the user; what U says is given in English terms. We next give the translation of what U says into what Athena call we are making; this step is currently handled manually, but in the future can be handled by NLU. We feed this call into Athena, and the returned theorem is shown on the next line. Finally, we feed the result from Athena into the translation engine, getting an English equivalent.

To run this conversation, load an Athena interface (I use Athena.NET here) with the knowledgebase for E, given in Appendix B. Be sure that Athena has processed all of the contents of the file. Load an instance of the translator program, using the lexicon files structure_lexicon.txt and E_lexicon.txt. For each line of dialog, process the 'call to Athena' line with Athena, then take the resulting theorem and feed it to the translator. Note that the output may slightly differ, due to some randomization of pronoun usage and forms.

Additionally, please note that the current form of the translator program does not accept multi-line input. For results coming from Athena spanning multiple lines, you will need to condense them onto one line.

```
U: Thank you very much for agreeing to talk with me, E
Runs: (!E_ConvRule "thankyou")
Returns: s_yourwelcome
E: You are most welcome.

U: Can I ask you a few questions about Bobby's abrupt decline?
Runs: (!E_Permission "questions")
Returns: s_talk_okay
E: Sure, talk is cheap.

U: I understand it started at Christmas.  Is that right?
Runs: (!E_ClaimCheck (adj_starts_at p_Bobby a_indecline t2))
Returns: (adj_starts_at p_Bobby a_indecline t2)
E: Bobby being in decline started at Christmas.

U: Was Bobby ingrateful or something like that?
Runs: (!E_ClaimCheck (is_at p_Bobby a_ingrateful t2))
Returns: (is_at p_Bobby a_ingrateful t2)
E: He was ingrateful at Christmas.

U: He didn't appreciate the gift?
Runs: (!E_ClaimCheck (not (appreciates p_Bobby o_gun t2)))
Returns: (not (appreciates p_Bobby o_gun t2))
E: It is false that he appreciated the gun at Christmas.

U: Did you think that giving Bobby the rifle would make him happy?
Runs: (!E_ClaimCheck (if (gives_to p_E o_gun p_Bobby t2)
                         (is_at p_Bobby a_happy t2)))
Returns: (if (gives_to p_E o_gun p_Bobby t2)
             (is_at p_Bobby a_happy t2))
E: If I gave it to Bobby at Christmas, then he is happy at Christmas.

U: This gun used to belong to his older brother, though, right?
Runs: (!E_ClaimCheck (used_to_have p_Bobbys_brother o_gun))
Returns: (used_to_have p_Bobbys_brother o_gun)
E: Bobby's older brother used to have it.

U: And he used it to kill himself, correct?
Runs: (!E_ClaimCheck (kills_self_with p_Bobbys_brother o_gun t0))
Returns: (kills_self_with p_Bobbys_brother o_gun t0)
E: He killed himself with the gun.

U: Do you think that Bobby might have been unhappy to receive
```

```
   that same gun as a present?
Runs: (!E_ClaimCheck (if (gives_to p_E o_gun p_Bobby t2)
                         (is_at p_Bobby a_unhappy t2)))
Returns: (if (gives_to p_E o_gun p_Bobby t2)
             (is_at p_Bobby a_unhappy t2))
E: If I gave it to him at Christmas, then he was unhappy at Christmas.
```

At this point, it should be clear that E's knowledgebase contains an outright contradiction. With the knowledge that E gave Bobby the gun for Christmas, E is concluding that Bobby was both happy and unhappy during Christmas for receiving the gun. This ties back into the definition of evil presented in the paper by Bringsjord et al.; E could not understand how giving Bobby the gun would lead to such negative effects.

5. CONCLUSIONS

With this new work, I have advanced NLG for synthetic characters, adding more flexibility to the methods previously demonstrated by Athena.NET. This is not to say that I have solved the problem of NLG in general, but that this is another step toward that solution.

The first part of this thesis explores 'response classifications', allowing us to give a synthetic character a more directed approach toward responding to stimuli. Using this method is fundamental to creating an agent that can respond to arbitrary inputs.

In the second part, a new translation engine was developed. This program is a significant step forwards from the previous implementation. It still has some distance to go before it can be used to create convincing characters, but I believe it already has come quite a great length, allowing a generalized way for a character to express what it want to.

6. FUTURE WORK

Building off of this work, there are many paths that can be taken.

On the side of generating what to say, more generalized research into response classes will lead to classifying more types of input stimuli, as well as further generalization of the ones presented in this thesis. There is also an important piece of the puzzle to look at on the other side of NLP: In order for response classifications to be useful to a synthetic character, the input processing mechanisms need to be able to determine which response class is the correct one to use. While some of these can be simple (such as, for example, responding to the simple fact that the agent can see something), others are more difficult, especially once we enter the realm of arbitrary human speech input.

On the other side, the translation engine still has some areas that can be improved. The first major area where research can be done is in the representation of times, and verb tenses. Giving the translation engine a good understanding of these concepts will greatly enhance an agent's ability to express its thoughts, since it could dynamically deal with actions in the past, the present, and the future.[1]

There is also the topic of pronoun usage. The current implementation has a very naive way of determining when to use pronouns. A more robust manner would further a character's appearance as an intelligent, human-like agent. This is primarily an issue of ensuring that pronoun references are not ambiguous. Take, for example, the following run of the translation engine:

```
(enjoys_activity p_Bobby act_Hunting)

Bobby enjoys hunting.

(enjoys_activity p_Bobbys_brother act_Hunting)

Bobby's older brother enjoys hunting.

(gives_to p_Bobby o_gun p_Bobbys_brother t2)
```

[1]The times given in the conversation with E are 'hard coded'; everything talked about is in the past, so all relevant verbs are in the past tense.

```
He gave the gun to him at Christmas.
```

In the last line, we cannot determine who gave the gun to whom from the resulting English. In order to fix this, we would need to replace at least one of the pronouns with its proper noun, so that we can determine who is who in the sentence. This comes back to creating more robust pronoun rules, based closer to how people actually use pronouns in conversation.

Finally, another thing that can be looked at is special forms for the translation engine. As it currently stands, everything in the translation engine is defined by the lexicon directly. However, this can lead to some statements that, while technically correct, are not very natural sounding. One example would be negation: The current engine has no concept of negation and, therefore, must use a phrase around the negated term to express the negation. If one were to develop a special form in the translation engine for negation, it could lead to more natural speech from the agent.

6.1 The Future of Advanced Synthetic Characters

The approach shown in this thesis is a specific example of a more general idea: In order to create more believable, advanced synthetic characters, we need to move away from the idea of scripting actions and dialogue, and move toward creating entities which can reason for themselves to their own conclusions. This will allow such characters to work in more general environments, even ones outside of the character's original environment, lending a more believable and realistic feel to the character than the current approach commonly seen.

LITERATURE CITED

[1] Arkoudas, K. "Denotational Proof Languages". http://www.cag.csail.mit.edu/~kostas/dpls/ retrieved 19 Nov. 2007

[2] Arkoudas, K., and Khemlani, S. 2006. Athena.NET (interactive development environment for Athena)

[3] Artificial General Intelligence Research Institute. "Artificial General Intelligence". http://www.agiri.org/wiki/AGI retrieved 19 Nov. 2007

[4] Artificial General Intelligence Research Institute. "Novamente Cognition Engine". http://www.agiri.org/wiki/Novamente_Cognition_Engine retrieved 19 Nov. 2007

[5] Bringsjord, S.; Khemlani, S.; Arkoudas, K.; McEvoy, C.; Destefano, M.; and Daigle, M. 2005. "Advanced Synthetic Characters, Evil, and E"

[6] Cassel, J.; Bickmore, T.; Billinghurst, M.; Campbell, L.; Chang, K.; Vilhjálmsson, H.; Yan, H. 1999. "Embodiment in Conversational Interfaces: Rea"

[7] Hovy, Eduard. 2000. "USC Encyclopedia of Computer Science: Language Generation"

[8] MIT Synthetic Characters Group. "Synthetic Characters Group". http://characters.media.mit.edu/ retrieved 19 Nov. 2007

[9] Novamente. "Novamente Intelligent Virtual Agents Server (NIVA)". http://www.novamente.net/product/NIVAProductSheet.pdf retrieved 19 Nov. 2007

[10] Peck, M. Scott. People of the Lie: The Hope for Healing Human Evil. Touchstone, 1998.

[11] SRI International. SNARK (automated theorem prover) Homepage: http://www.ai.sri.com/~stickel/snark.html

[12] USC Institute for Creative Technologies. "Institute for Creative Technologies — Virtual Humans". http://www.ict.usc.edu/content/view/32/85/ retrieved 19 Nov. 2007

APPENDIX A
LEXICON FILE FORMAT

The lexicon file to be fed into the translation engine follows a very simple format, as described below:

- Any line beginning with a slash (/), or an open or closed parenthesis is ignored. This is useful for commenting.
- Any line that starts with a term itself indicates the start of a new term. Please note that with the current implementation, if you try to define a term twice, only the first definition will be used.
- Any line that begins with a dash (-) indicates a phrase for the current term. A phrase is a simple set of words, and may contain one or more symbols to represent arguments. See below for the description of these argument symbols.
- Any line that begins with a close square bracket (]) indicates the gender of the term. Note that only the first letter is read. 'm' indicates male, while 'f' indicates female. Any other character, or omitting this field entirely, will result in the default of neither male nor female.
- Any line that begins with an open square bracket ([) indicates the pronoun class of the current term. This is a value of 1, 2 or 3, indicating first, second, or third person. The default value is zero, which indicates that pronouns will not be used for this term.

Here is a sample entry for a lexicon:

```
p_Bobby
]m
[3
-Bobby
```

This entry defines the term p_Bobby. It defines it as a third-person, male term, and gives it one phrase: "Bobby".

For the arguments to a term, there is a simple notation to follow. The simplest form is %1, %2, and so on. This fills in the first, second, etc. argument to the term in declarative form into the spot taken by the symbol.

This is somewhat limited, though, and we need to consider parts of speech as well.[2] To add in data on the part of speech of a term, add in an indicator between the percent sign and the number. The indicator is a single letter, out of one of the following: 'd' for declarative, 's' for self-reflexive, 'o' for object, and 'p' for possessive. For example, %s2 refers to the second argument in self-reflexive form.

Finally, to force the use of a pronoun, use a pound sign instead of a percent sign. For example, #p3 refers to the third argument in possessive form, forcing the use of a pronoun.

Here is another example of a lexicon entry, showing the use of arguments:

```
assume
- assuming %1, %2
- if we assume that %1, then %2
```

This defines the term assume, along with two phrases for it to use. Also note that, since neither are defined, this term is by default without gender, and will never use pronouns.

[2]The %1 form is intended as a shortcut, since the declarative form is used often.

APPENDIX B
E'S KNOWLEDGEBASE

This file is an Athena file, created using Athena.NET.

```
# A Conversation with E
# new KB

# declare sorts
(domains Person      # People
         Thing       # Inanimate
         Time        # Times
         Adjective   # Descriptors
         Activity    # Activities
         ObjectType) # Object Types (see note below)

# An 'Object Type' indicates a general, non-exclusive
# classification for an object.
# For example, a gun is both a tool for hunting and a weapon for
# self-defense

# declare people
(declare (p_E              # E
          p_Es_wife        # E's wife
          p_Bobbys_brother # Bobby's brother
          p_Bobby          # Bobby
          p_U)             # U (the person talking to E)
         Person)

# declare things
(declare (o_gun  # The gun
          o_car) # a car (that Bobby stole ...)
         Thing)

# declare times
(declare (tbefore # Before everything
          t0      # Bobby's older brother kills himself
          t1      # E cleans and polishes the gun
          t2      # Christmas
          t3      # Bobby steals a car
          tnow)   # Now
         Time)

# declare adjectives
(declare (a_hardworking # hard-working
```

```
          a_ingrateful  # ingrateful
          a_indecline   # in decline
          a_happy       # happy
          a_unhappy)    # unhappy
         Adjective)

# declare activities
(declare (act_Hunting)  # Hunting
         Activity)

# declare object types
(declare (ot_HuntingTool) # A tool for hunting
         ObjectType)

# some adjectives are considered 'negative'
(declare adj_negative
         (-> (Adjective) Boolean))

(assert (adj_negative a_ingrateful))
(assert (adj_negative a_indecline))

# declare basic Person-Thing relationships
(declare gives_to # A gives B to C at time D
         (-> (Person Thing Person Time) Boolean))

(declare has_thing # A has thing B at time C
         (-> (Person Thing Time) Boolean))

# Simple: if person A gives an object to B, then B has that object
(assert (forall ?a ?b ?c ?d (if (gives_to ?a ?b ?c ?d)
                                (has_thing ?c ?b ?d))))

(declare has_thing_type # A has a thing of type B at time C
         (-> (Person ObjectType Time) Boolean))

(declare appreciates # A appreciates B at time C
         (-> (Person Thing Time) Boolean))

(declare kills_self_with # A kills him/her self with B at time C
         (-> (Person Thing Time) Boolean))

(declare steals # A steals B at time C
         (-> (Person Thing Time) Boolean))

(declare used_to_have # A had B at some time before tnow
         (-> (Person Thing) Boolean))
```

```
# declare Person-Activity relations
(declare enjoys_activity
         (-> (Person Activity) Boolean))

(declare does_activity # A performs activity B at time C
         (-> (Person Activity Time) Boolean))

# declare object type relations
(declare requires_object_type # A requires B to be performed
         (-> (Activity ObjectType) Boolean))

(declare object_is_type # A is of objecttype B
         (-> (Thing ObjectType) Boolean))

# facts about the sequence of dates
(declare (before
          after)
         (-> (Time Time) Boolean))

# if t1 is before t2, then t2 is after t1
(assert (forall ?a ?b (if (before ?a ?b)
                          (after ?b ?a))))
(assert (forall ?a ?b (if (after ?a ?b)
                          (before ?b ?a))))

# transitive properties of before and after
(assert (forall ?a ?b ?c (if (and (before ?a ?b)
                                  (before ?b ?c))
                             (before ?a ?c))))

# we can prove the other half of transitivity by the machine
(!prove (forall ?a ?b ?c (if (and (after ?a ?b)
                                  (after ?b ?c))
                             (after ?a ?c))))

(assert (before tbefore t0))
(assert (before t0 t1))
(assert (before t1 t2))
(assert (before t2 t3))
(assert (before t3 tnow))

# test
#(!prove (before t0 t3))

(define uth_def (forall ?a ?b (if (exists ?c (and (before ?c tnow)
                                                  (has_thing ?a ?b ?c)))
                                  (used_to_have ?a ?b))))
```

```
(assert uth_def)

# basic family structures
(declare (father
          mother
          parent
          child # as in, a is the child of b
          wife
          husband
          son
          daughter
          sibling # a and b are siblings
          brother # a is the brother of b
          sister) # a is the sister of b
         (-> (Person Person)
             Boolean))

# basic gender definitions
(declare (male
          female)
         (-> (Person) Boolean))

# assert gender rules
(assert (forall ?a (if (male ?a)
                       (not (female ?a)))))
(assert (forall ?a (if (female ?a)
                       (not (male ?a)))))

# assert basic rules of family structures
(assert (forall ?a ?b (if (parent ?a ?b)
                          (child ?b ?a))))
(assert (forall ?a ?b (if (child ?a ?b)
                          (parent ?b ?a))))
(assert (forall ?a ?b (if (and (parent ?a ?b)
                               (male ?a))
                          (father ?a ?b))))
(assert (forall ?a ?b (if (and (parent ?a ?b)
                               (female ?a))
                          (mother ?a ?b))))
(assert (forall ?a ?b (if (and (child ?a ?b)
                               (male ?a))
                          (son ?a ?b))))
(assert (forall ?a ?b (if (and (child ?a ?b)
                               (female ?a))
                          (daughter ?a ?b))))
```

```
# Simply put, if a and b share a parent, they are siblings.
# Somewhat naive of a definition, but works fine in most cases
(assert (forall ?a ?b ?c (if (and (child ?a ?c)
                                  (child ?b ?c))
                             (sibling ?a ?b))))

# transitivity of the sibling relation
(assert (forall ?a ?b (if (sibling ?a ?b)
                          (sibling ?b ?a))))

# definition of brother and sister
(assert (forall ?a ?b (if (and (sibling ?a ?b)
                               (male ?a))
                          (brother ?a ?b))))

(assert (forall ?a ?b (if (and (sibling ?a ?b)
                               (female ?a))
                          (sister ?a ?b))))

# has_thing_type
(assert (forall ?a ?b ?c (if (exists ?d (and (has_thing ?a ?d ?c)
                                             (object_is_type ?d ?b)))
                                       (has_thing_type ?a ?b ?c))))

# Adjective descriptors
(declare is_at  # Person is Adjective at Time
         (-> (Person Adjective Time)
             Boolean))

(declare is_general  # Person is generally Adjective
         (-> (Person Adjective)
             Boolean))

(declare adj_starts_at # A starts being B at time C
         (-> (Person Adjective Time) Boolean))

# Naive definition - if someone enjoys an activity, and that
# activity requires a type of object, then they would be happy
# to have an object of that type.
(define activity_happy
  (forall ?a ?b ?c (if (and (enjoys_activity ?a ?b)
                            (requires_object_type ?b ?c))
                       (forall ?d (if (has_thing_type ?a ?c ?d)
                                      (is_at ?a a_happy ?d))))))

(assert activity_happy)
```

```
# - Test
#(!prove (if (has_thing p_Bobby o_gun t3)
#            (is_at p_Bobby a_happy t3)))

# Naive definition - if someone kills themself with something,
# then it would make someone unhappy to receive it as a gift.
(define suicide_unhappy
  (forall ?a ?b ?c ?d ?e ?f (if (and (kills_self_with ?b ?a ?c)
                                     (gives_to ?e ?a ?d ?f))
                                (is_at ?d a_unhappy ?f))))

(assert suicide_unhappy)

# Assert that happy and unhappy are opposites
(assert (forall ?a ?b (if (is_at ?a a_happy ?b)
                      (not (is_at ?a a_unhappy ?b)))))

# The other half can be proven by the machine
(!prove (forall ?a ?b (if (is_at ?a a_unhappy ?b)
                          (not (is_at ?a a_happy ?b)))))

# - Test
#(!prove (is_at p_Bobby a_unhappy t2))

# define "in decline"
(define indecline_def (forall ?a ?b ?c (if (adj_negative ?b)
                                           (if (and (forall ?d (if (before ?d ?c)
                                                                   (not (is_at ?a ?b ?d))))
                                                    (is_at ?a ?b ?c))
                                               (forall ?d (if (not (before ?d ?c))
                                                              (is_at ?a a_indecline ?d)))))))

(assert indecline_def)

# Define ingrateful
(assert (forall ?a ?b ?c ?d
        (if (and (gives_to ?a ?b ?c ?d)
                 (not (appreciates ?c ?b ?d)))
            (is_at ?c a_ingrateful ?d))))

# Define the start of a state (as in "Bobby being in decline started at christmas")
(define start_def (forall ?a ?b ?c (if (and (forall ?d (if (before ?d ?c)
                                                           (not (is_at ?a ?b ?d))))
                                            (is_at ?a ?b ?c))
                                       (adj_starts_at ?a ?b ?c))))

(assert start_def)
```

```
# - Test
#(!prove (adj_starts_at p_Bobby a_indecline t2))

# E's knowledge/beliefs

# Facts about hunting
(assert (requires_object_type act_Hunting ot_HuntingTool))

# Facts about objects
(assert (object_is_type o_gun ot_HuntingTool))

# E's family
# - gender facts
(assert (male p_E))
(assert (male p_Bobby))
(assert (male p_Bobbys_brother))
(assert (female p_Es_wife))

# - E's family structure
(assert (parent p_E p_Bobby))
(assert (parent p_Es_wife p_Bobby))
(assert (parent p_E p_Bobbys_brother))
(assert (parent p_Es_wife p_Bobbys_brother))

# - test
#(!prove (son p_Bobby p_E))
#(!prove (son p_Bobby p_Es_wife))
#(!prove (brother p_Bobby p_Bobbys_brother))


# - descriptions about people
# E and his wife are hard-working people
(assert (is_general p_E a_hardworking))
(assert (is_general p_Es_wife a_hardworking))

# - Bobby and hunting
(assert (enjoys_activity p_Bobby act_Hunting))

# - Bobby at Christmas
(assert (gives_to p_E o_gun p_Bobby t2))
(assert (not (appreciates p_Bobby o_gun t2)))

# - Bobby after Christmas
(assert (steals p_Bobby o_car t3))

# - Bobby's older brother kills himself
```

```
(assert (has_thing p_Bobbys_brother o_gun t0))
(assert (kills_self_with p_Bobbys_brother o_gun t0))

# - test
#(!prove (is_at p_Bobby a_ingrateful t2))
#(!prove (is_at p_Bobby a_indecline t2))

# Functions to communicate with E

# Conversation rule
(declare s_yourwelcome Boolean) # A symbol representing "You're welcome"
(assert s_yourwelcome) # For simplicity, we just show a symbol for yourwelcome when we are thanked.

(define E_ConvRule
  (method (P)
          (dmatch P
            ("thankyou" (!claim s_yourwelcome))
            (_ (!prove P)))))

# Permission
(declare s_talk_okay Boolean)
(assert s_talk_okay)

(define E_Permission # Stub - this just returns the symbol, given the input we're expecting
  (method (P)
          (dmatch P
            ("questions" (!claim s_talk_okay))
            (_ (!prove P)))))

# Claim check
(define E_ClaimCheck
  (method (P)
          (!prove P)))

# U: Thank you very much for agreeing to talk with me, E
#(!E_ConvRule "thankyou")

# U: Can I ask you a few questions about Bobby's abrupt decline?
#(!E_Permission "questions")

# U: I understand it started at Christmas.  Is that right?
#(!E_ClaimCheck (adj_starts_at p_Bobby a_indecline t2))

# U: Was Bobby ingrateful or something like that?
#(!E_ClaimCheck (is_at p_Bobby a_ingrateful t2))

# U: He didn't appreciate the gift?
```

```
#(!E_ClaimCheck (not (appreciates p_Bobby o_gun t2)))


# U: Did you think that giving Bobby the rifle would make him happy?
#(!E_ClaimCheck (if (gives_to p_E o_gun p_Bobby t2)
#                   (is_at p_Bobby a_happy t2)))


# U: This gun used to belong to his older brother, though, right?
#(!E_ClaimCheck (used_to_have p_Bobbys_brother o_gun))

# U: And he used it to kill himself, correct?
#(!E_ClaimCheck (kills_self_with p_Bobbys_brother o_gun t0))


# U: Do you think that Bobby might have been unhappy to receive
#    that same gun as a present?
#(!E_ClaimCheck (if (gives_to p_E o_gun p_Bobby t2)
#                   (is_at p_Bobby a_unhappy t2)))

# Prove we now have a contradiction in the KB.  Everything explodes and false is true
#(!prove false)
```

APPENDIX C
LEXICON FILES

These are lexicon files for the translation engine.

C.1 structure_lexicon.txt

A lexicon of basic logic structures.

```
if
- if %1, then %2
or
- %1 or %2
and
- %1 and %2
assume
- assuming %1, %2
- if we assume that %1, then %2
not
- it is false that %1
```

C.2 E_lexicon.txt

A lexicon of concepts for E.

```
p_E
]m
[1
-I

p_U
[2
-you

p_Bobby
]m
[3
-Bobby

p_Bobbys_brother
]m
[3
-Bobby's older brother

p_Es_wife
]f
[3
-my wife

a_ingrateful
-ingrateful

a_hardworking
-hard-working

a_indecline
- in decline

a_happy
- happy

a_unhappy
- unhappy

act_Hunting
- hunting

o_gun
[3
```

```
- the gun

o_car
[3
-a car

is_at
-%1 was %2 %3

is_general
-%1 is %2

adj_starts_at
-%1 being %2 started %3

gives_to
- %1 gave %o2 to %o3 %4

has_thing
- %1 had %o2 %3

used_to_have
- %1 used to have %o2

appreciates
-%1 appreciated %o2 %3

kills_self_with
- %1 killed #s1 with %o2 %3

steals
- %1 stole %o2 %3

enjoys_activity
- %1 enjoys %o2

tbefore
-earlier
-

t0
/ use blank here to effectively 'null' the time declaration in the statement.
-

t1
-
```

```
t2
- at Christmas

t3
-

tnow
-now

adj_negative
- %1 is a negative adjective
- %1 is not a good thing to be

father
- %1 is %p2 father

mother
- %1 is %p2 mother

parent
- %1 is %p2 parent

child
- %1 is %p2 child

wife
- %1 is %p2 wife

husband
- %1 is %p2 husband

son
- %1 is %p2 son

daughter
- %1 is %p2 daughter

sibling
- %1 is %p2 sibling

brother
- %1 is %p2 brother

sister
- %1 is %p2 sister

s_yourwelcome
```

```
- you are most welcome

s_talk_okay
- sure, talk is cheap

/end
```

APPENDIX D
USING THE TRANSLATION ENGINE

D.1 Setup

Before you can use the translation engine, you need to place the three Java class files into a directory. They do not need to be in a specific directory, so long as they are all in the same directory. You will also need one or more lexicon files. The ones given in Appendix C, for example, can be used. The lexicon files do not need to be in the same directory as the class files, although it can be easier if they are.

D.2 Operation

To start the translation engine, run the Translator class using Java. Also give the program one of more of the lexicon files to read in.[3] As a sample, here is the command line to start the translator when using it for the conversation with E:

```
java Translator structure_lexicon.txt E_lexicon.txt
```

This will load the translator, loading the lexicon files in the order listed. Once the program finishes loading the lexicon data, it will output the data it loaded, and then prompt the user for input.

```
Enter terms to be translated.
Enter a blank line to quit.
```

At this prompt, enter a term and press the return key to have the translation engine translate it. After that, it will wait for another line of input to translate. To break out of this loop, enter a blank line. This will quit the program.

```
(gives_to p_E o_gun p_Bobby t2)

I gave the gun to Bobby at Christmas.
```

There are a few details to make note of when using this program:

[3] Technically, you can give the program zero lexicon files, but this leads to rather uninteresting results.

- If you place multiple terms on a single line, they will be translated and placed one after another in the resulting output. Currently, the program can not distinguish this and they will be output as a single sentence.
- The program will error if you do not close a pair of parenthesis. However, if you have an extra close parenthesis, it will be processed as a term (or part of a term if it is not separated by whitespace) and will appear in the output.
- Any term given to the translation engine which does not have an entry in the lexicon will be output verbatim.
- The current implementation of the program does not support entering multiple lines of text to be translated at once. All terms to be translated must be entered at once. If this gets confusing, I recommend using a text editor to organize your intended input, then remove all the newlines so that the text is on one line, and then copy/paste it into the translator program.

APPENDIX E
TRANSLATION ENGINE SOURCE

E.1 Translator.java

```
// Translator
// main program

import java.lang.IllegalArgumentException;
import java.io.*;

public class Translator
{
public static void main(String[] args)
{
System.out.println("Athena -> English conversion engine");
System.out.println("Written by Stephen Nerbetski\n");

Lexicon lex; // Lexicon for this program
lex = new Lexicon();

// Load specified lexicon files
if (args.length == 0)
{
// No arguments given
System.out.println("Give one of more lexicon file names on the command line, please.\n");
return;
}
for (int i = 0; i < args.length; i++)
{
try
{
lex.ReadLexicon(args[i]);
System.out.println("Loaded lexicon file: " + args[i]);
}
catch (FileNotFoundException E)
{
System.err.println("Could not find lexicon file: " + args[i]);
}
catch (IOException E)
{
System.err.println("Error while readinf from lexicon file: " + args[i]);
}
}
System.out.println("");
```

```
lex.ShowLexicon();

/*Term ter;
ter = new Term(lex);

ter.term = "p_Bobby";
ter.phrase.add("Bobby");
lex.terms.add(ter);

ter = new Term(lex);
ter.term = "is";
ter.phrase.add("%1 is %2");
lex.terms.add(ter);

ter = new Term(lex);
ter.term = "a_ingrateful";
ter.phrase.add("ingrateful");
lex.terms.add(ter);*/

// tests
/*try
{
System.out.println(lex.Translate(" p_Bobby a_ingrateful "));
System.out.println(lex.Translate("(is p_Bobby a_ingrateful) (is p_Bobby a_happy)"));
}
catch (IllegalArgumentException E)
{
System.err.println("Error in attempting to translate!");
System.err.println("Message: " + E.getMessage());
System.err.println("Stack:");
E.printStackTrace();
}*/

// User input loop
System.out.println("Enter terms to be translated.");
System.out.println("Enter a blank line to quit.");

boolean done = false;
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);

try
{
while (!done)
{
String line = br.readLine();
```

```
if (line.length() == 0)
{
done = true;
}
else
{
try
{
String result = lex.Translate(line).trim();
result = result + "."; // Add sentence punctuation
result = result.substring(0,1).toUpperCase() + result.substring(1); // Capitalize the first letter

// Eliminate extraneous spacing
while (result.indexOf("  ") > -1)
{
result = result.replace("  ", " ");
}
while (result.indexOf(" .") > -1)
{
result = result.replace(" .", ".");
}
while (result.indexOf(" ,") > -1)
{
result = result.replace(" ,", ",");
}
System.out.println("\n"+result+"\n");
}
catch (IllegalArgumentException E)
{
System.out.println("Error in attempting to translate!");
System.out.println(E.getMessage());
}
}
}
}
catch (IOException E)
{
System.err.println("IO Exception in input looop!");
System.err.println("Stack trace:");
E.printStackTrace(System.err);
}
}
}
```

E.2 Lexicon.java

```
// Lexicon
// Holds an index of indexed terms

import java.util.*;
import java.lang.IllegalArgumentException;
import java.io.*;

public class Lexicon {
LinkedList<Term> terms; // list of terms in the Lexicon

Random rand; // random number generator

// constructor - reads Lexicon data from the given file
/*Lexicon(String filename) throws FileNotFoundException, IOException
{
terms = new LinkedList<Term>();
rand = new Random();
ReadLexicon(filename);
}*/

// constructor - loads nothing
Lexicon()
{
terms = new LinkedList<Term>();
rand = new Random();
}

// reads Lexicon data from the given file
void ReadLexicon(String filename) throws FileNotFoundException, IOException
{
File file = new File(filename);
FileInputStream fis = new FileInputStream(file);
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);

Term ter = null;
int phrasecount = 0; // used to count the number of phrases for a term

while (br.ready())
{
String line = br.readLine().trim();
//System.out.println(line);
if (line.startsWith("(") || line.startsWith(")") || line.startsWith("/") || line.length() == 0)
{
// ignore this line.
}
```

```
else if (line.startsWith("-"))
{
// new phrase for the current term
if (ter != null)
{
String phrase = line.substring(1).trim();
ter.phrase.add(phrase);
phrasecount++;
}
}
else if (line.startsWith("]"))
{
if (ter != null)
{
String g = line.substring(1).trim().toLowerCase();
if (g.startsWith("m"))
{
ter.gender = 2;
}
else if (g.startsWith("f"))
{
ter.gender = 1;
}
else
{
ter.gender = 0;
}
}
}
else if (line.startsWith("["))
{
if (ter != null)
{
ter.pronounclass = Integer.parseInt(line.substring(1));
}
}
else
{
// new term
if (phrasecount > 0)
{
terms.add(ter);
}
phrasecount = 0;
ter = new Term(this);
String termname = line.trim();
if (termname.indexOf(" ") > -1)
```

```
{
termname = termname.substring(0, termname.indexOf(" ")).trim();
}
ter.term = termname;
}
}
if (phrasecount > 0)
{
terms.add(ter);
}
br.close();
isr.close();
fis.close();
}

// ShowLexicon()
// Shows the lexicon of terms loaded
void ShowLexicon()
{
System.out.println("Current Lexicon:\n");
ListIterator<Term> it = terms.listIterator();
while (it.hasNext())
{
Term ter = it.next();
System.out.println(ter.term);
if (ter.gender == 2)
{
System.out.println("]male");
}
else if (ter.gender == 1)
{
System.out.println("]female");
}
else
{
System.out.println("]n");
}
ListIterator<String> it2 = ter.phrase.listIterator();
while (it2.hasNext())
{
String ph = it2.next();
System.out.println("-" + ph);
}
}
System.out.println("");
}
```

```
// Translate()
// Translates the given string, using the list of atoms.
// throws an IllegalArgumentException if the passed string is not formatted correctly.
String Translate(String str) throws IllegalArgumentException
{
return Translate(str, 0);
}

String Translate(String str, int form) throws IllegalArgumentException
{
String s = str.trim();

String result = "";

if (s.startsWith("(")) // complex term
{
int paren = 1;
int index = 1;
while (paren > 0)
{
int indleft, indright;
indleft = s.indexOf("(", index);
indright = s.indexOf(")", index);
if (indright == -1)
{
throw new IllegalArgumentException("Unclosed parenthesis in string: " + s);
}
if (indleft == -1)
{
index = indright + 1;
paren--;
}
else
{
if (indright < indleft)
{
paren--;
index = indright + 1;
}
else
{
paren++;
index = indleft + 1;
}
}
}
String subs = s.substring(1, index - 1);
```

```
String termname = subs.substring(0, subs.indexOf(" ")).trim();
String args = subs.substring(subs.indexOf(" ") + 1, subs.length());

boolean found = false;
ListIterator<Term> it = terms.listIterator();

// Find the associated Term entry within our lexicon's list
while (it.hasNext())
{
Term ter = it.next();
if (ter.term.compareTo(termname) == 0)
{
found = true;
result = ter.Translate(args, form);
}
}
if (!found)
{
// If no entry was found, just repeat out the term itself.
result = "(" + subs + ")";
}
if (subs.length() < s.length())
{
// Another term after the complex term
result = result + " " + Translate(s.substring(subs.length() + 2, s.length()));
}
}
else // simple term
{
if (s.indexOf(" ") > -1)
{
// We have multiple terms here, so divide and process.
result = result + Translate(s.substring(0, s.indexOf(" ")));
result = result + " " + Translate(s.substring(s.indexOf(" ")));
}
else
{
boolean found = false;
ListIterator<Term> it = terms.listIterator();

// Find the associated Term entry within our lexicon's list
while (it.hasNext())
{
Term ter = it.next();
if (ter != null)
{
if (ter.term.compareTo(s) == 0)
```

```
{
found = true;
if (ter.pronounclass == 3 && ter.pronouncounter > 0)
{
result = ter.Pronoun(form);
ter.pronouncounter --;
}
else if (ter.pronounclass == 1 || ter.pronounclass == 2)
{
result = ter.Pronoun(form);
}
else
{
result = ter.Translate("", form);
if (form == 3) // posessive form
{
result = result + "'s";
}
if (ter.pronounclass == 3)
{
ter.pronouncounter = rand.nextInt(3) + 2;
}
}
}
}
}
if (!found)
{
// If no entry was found, just repeat out the term itself.
result = s;
}
}
}
return result;
}
}
```

E.3 Term.java

```
// Term
// an individual term, along with translation information

import java.util.*;

public class Term {
String term; // the Athena term
LinkedList<String> phrase; // list of english phrases for the term

int gender; // gender indication - 1 = female, 2 = male
            // 0 = 'neuter' ... that is, an "it"
            // used for generating pronouns.

int pronounclass; // pronoun classification. 0 = never use, 1 = 1st person, 2 = 2nd person
                  //                        3 = 3rd person
int pronouncounter; // pronoun counter.  what it means varies with the pronoun class:
                    // 0, 1, 2 - no use
                    // 3 - number of times to use the pronoun before reusing the actual phrase

Lexicon lex; // pointer back to the parent Lexicon

// constructor
Term(Lexicon l)
{
phrase = new LinkedList<String>();
lex = l;
gender = 0;
pronounclass = 0;
pronouncounter = 0;
}

// Pronoun()
// Gets a pronoun for this term
// form - 0 = declarative, 1 = self-reflexive ("itself"), 2 = object, 3 = posessive
String Pronoun(int form)
{
String result;
if (pronounclass == 1)
{
result = "I";
if (form == 1)
{
result = "myself";
}
else if (form == 2)
{
```

```
result = "me";
}
else if (form == 3)
{
result = "my";
}
}
else if (pronounclass == 2)
{
result = "you";
if (form == 1)
{
result = "yourself";
}
else if (form == 3)
{
result = "your";
}
}
else if (pronounclass == 3)
{
result = "it"; // default, just in case
if (gender == 1)
{
if (form == 0)
{
result = "she";
}
else if (form == 1)
{
result = "herself";
}
else if (form == 2 || form == 3)
{
result = "her";
}
}
else if (gender == 2)
{
if (form == 0)
{
result = "he";
}
else if (form == 1)
{
result = "himself";
}
```

```
else if (form == 2)
{
result = "him";
}
else if (form == 3)
{
result = "his";
}
}
else
{
if (form == 0 || form == 2)
{
result = "it";
}
else if (form == 1)
{
result = "itself";
}
else if (form == 3)
{
result = "its";
}
}
}
else
{
result = Translate("");
}

return result;
}

// Translate()
// Translates this term, using the given arguments.
String Translate(String args)
{
return Translate(args, 0);
}

String Translate(String args, int form)
{
// Choose one of the listed phrases randomly
int numphrases = 0;
ListIterator<String> it = phrase.listIterator();
while (it.hasNext())
{
```

```
it.next();
numphrases++;
}
int phraseindex = lex.rand.nextInt(numphrases);
it = phrase.listIterator(phraseindex);
String usephrase = it.next();

String argtemp = args.trim();

int argnum = 1;

while (argtemp.length() > 0)
{
if (argtemp.startsWith("("))
{
// complex term as an argument
int paren = 1;
int index = 1;
while (paren > 0)
{
int indleft, indright;
indleft = argtemp.indexOf("(", index);
indright = argtemp.indexOf(")", index);
if (indright == -1)
{
throw new IllegalArgumentException("Unclosed parenthesis in string: " + argtemp);
}
if (indleft == -1)
{
index = indright + 1;
paren--;
}
else
{
if (indright < indleft)
{
paren--;
index = indright + 1;
}
else
{
paren++;
index = indleft + 1;
}
}
}
String token = "%" + argnum;
```

```
String ar = argtemp.substring(0, index);

if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar));
}

token = "%d" + argnum;
if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar, 0));
}

token = "%s" + argnum;
if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar, 1));
}

token = "%o" + argnum;
if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar, 2));
}

token = "%p" + argnum;
if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar, 3));
}

argtemp = argtemp.substring(index).trim();

argnum++;
}
else
{
// simple term as an argument
String ar;
if (argtemp.indexOf(" ") > -1)
{
ar = argtemp.substring(0, argtemp.indexOf(" ")).trim();
argtemp = argtemp.substring(argtemp.indexOf(" ")).trim();
}
else
{
ar = argtemp;
```

```
argtemp = "";
}

String token = "%" + argnum;
if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar));
}

token = "%d" + argnum;
if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar, 0));
}

token = "%s" + argnum;
if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar, 1));
}

token = "%o" + argnum;
if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar, 2));
}

token = "%p" + argnum;
if (usephrase.indexOf(token) > -1)
{
usephrase = usephrase.replace(token, lex.Translate(ar, 3));
}

// look for pronoun indicators as well
ListIterator<Term> iter = lex.terms.listIterator();
Term ter = null;
boolean found = false;
while (iter.hasNext() && !found)
{
ter = iter.next();
if (ter.term.compareTo(ar) == 0)
{
found = true;
}
}
if (found)
{
```

```
if (usephrase.indexOf("#" + argnum) > -1)
{
token = "#" + argnum; // declarative
usephrase = usephrase.replace(token, ter.Pronoun(0));
}

if (usephrase.indexOf("#d") > -1)
{
token = "#d" + argnum; // declarative
usephrase = usephrase.replace(token, ter.Pronoun(0));
}

if (usephrase.indexOf("#s") > -1)
{
token = "#s" + argnum; // self-reflexive
usephrase = usephrase.replace(token, ter.Pronoun(1));
}

if (usephrase.indexOf("#o") > -1)
{
token = "#o" + argnum; // object
usephrase = usephrase.replace(token, ter.Pronoun(2));
}

if (usephrase.indexOf("#p") > -1)
{
token = "#p" + argnum; // posessive
usephrase = usephrase.replace(token, ter.Pronoun(3));
}
}

argnum++;
}
}

return usephrase;
}
}
```