

Can Agent-Agent Simulation of Conversation be Extended to Create Literary Works?

Joaquin A. Aguilar A.

Computer Science Department

University of Texas at El Paso

APPROVED:

David G. Novick, Ph.D., Associate Provost and
AT&T Distinguished Professor of Computer
Science

Nigel Ward, Ph.D.

Ezra Cappell, Ph.D.

ACKNOWLEDGMENTS

This work could not have been possible without the valuable support I have received from my family and the guidance provided by my advisor during the course of this endeavor. Thank you.

ABSTRACT

LUKIS TALES is an agent-based implementation of a story-generation system. This thesis addresses whether agent-agent simulation of conversation can be extended to create literary works. Agents were created using a symbolic cognitive architecture named Soar. The agents were then able to pass messages to each other by means of a program written in C++ that uses the Soar markup language. The thesis also introduces criteria by which stories created by story-generation systems can be evaluated.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	2
ABSTRACT	3
TABLE OF CONTENTS	4
LIST OF FIGURES	6
Chapter	
1. STORYTELLING AND CREATIVITY	7
1.1. Creativity and computers	8
1.2. Literary works and agents	8
1.3. Creating a story generation system	8
1.4. Improving story generation systems	8
2. SURVEY OF RELATED WORK	10
2.1. TALE-SPIN	11
2.2. MISTREL	12
2.3. BRUTUS	13
2.4. MEXICA	15
2.5. VIRTUAL STORYTELLER	16
2.6. Summary	17
3. AGENT CONVERSATION SYSTEMS	19
3.1. Speech-act theory	19
3.2. Mutual model of conversation	20
3.3. Collaborative storytelling systems	20
3.4. Conclusion	21
4. LUKIS TALES	22
4.1. Agents as characters	22
4.2. Agent architecture	24
4.3. Interacting agents	26
4.4. Summary	26
5. EVALUATION OF LUKIS TALES IN TERMS OF ELEMENTS OF LITERARY WORKS	27
5.1. Drama	27
5.2. Freytag's triangle	28
5.3. Plot	28
5.4. Creating characters	29

5.5. Incident	30
5.6. Summary	31
6. CONCLUSION and FUTURE WORK	32
6.1. Conclusion	32
6.2. Future work	32
LIST OF REFERENCES	34
APPENDIX A – Sample conversations	36
APPENDIX B – Selected productions written in Soar	43
APPENDIX C – C++ application that uses the Soar markup language	54

LIST OF FIGURES

FIGURE 1. Internal representation of an agent.

CHAPTER 1

STORYTELLING AND CREATIVITY

This thesis addresses whether agent-agent simulation of conversation can be extended to create literary works. To that end, I developed a story-generation system, LUKIS TALES, whose goal is to have agents create a conversation, which can then be considered a literary work. The name LUKIS TALES is a reference to Loki, a mythical being from Norse mythology. The system was named after him because he represents war and comedy, which are two elements commonly found in literary works. The thesis also addresses related topics such as whether a computer can be creative, what it takes to create a story-generation system, how agents can be used to create literary works, and how to improve story-generation systems.

Creativity and computers

People who have done research in computer creativity, such as Bringsjord (2000), agree on Boden's (2004) definition of creativity. Boden, a research professor of cognitive science at Sussex University, defines creativity as “the ability to come up with ideas or artefacts that are *new, surprising, and valuable.*” (2004, p. 1). Boden then gives meanings to the terms *new, surprising, and valuable*. *New* can mean either P-creativity or H-creativity. P-creativity is when a person discovers something which he or she did not know without knowing that somebody else had already discovered it. H-creativity is when a person discovers something that nobody else had discovered before. *Surprising* can have three different meanings. The first is an idea that is unfamiliar or unlikely such as a comparison between a person and an animal. The second is an idea that is unexpected such as a move in a game of Go, which you would not have made before. The third is an idea that seems impossible, such as a computer that is conscious of itself and its environment. *Valuable*, however, has no meaning because different people attribute different values to different ideas.

The current state of artificial intelligence does not provide algorithms able to satisfy Boden's definition. However, whether computers may one day be creative or not is still open to discussion. Story-generation systems, described in Chapter Two, are created instead to appear creative. LUKIS TALES was created with this intention in mind. People find conversations created by LUKIS TALES creative the first time they see them. But afterwards, conversations become repetitive and boring because of the limited domain of the conversations.

Creating a story-generation system

Several approaches have been taken to create story-generation systems. These approaches have similarities such as using rule engines to control the story generation, translating the story to natural language after it has been created, and using a small amount of knowledge about a specific domain. They differ mainly in the theory they implement to create the stories. For example, some researchers believe that creativity is a problem-solving process and implement their system that way, while others disagree with this approach and opt to take another one. Chapter Three describes the approach I used to develop LUKIS TALES.

Literary works and agents

LUKIS TALES has agents interact with each other to create conversations, which when taken as a whole can be considered literary works. Literary works share some common elements. By evaluating LUKIS TALES in terms of these elements it is possible to consider conversations created by the system literary works. The evaluation of LUKIS TALES in terms of these elements, along with a description of them, can be found in Chapter Four.

Improving story-generation systems

To create interesting and compelling stories, story-generation systems must have a large corpus of knowledge about the real world and must be able to reason about it, which is not the

case in current story-generation systems. They must also implement a theory of writing that resembles the way humans create stories. Chapter Five addresses these issues by commenting on one recent project that incorporates a large corpus of knowledge along with ways to reason about this knowledge and by presenting my views on the challenges of developing a theory of writing.

CHAPTER 2

SURVEY OF RELATED WORK

Artificial intelligence has, in large part, focused on replicating the way humans act and think. Story-generation systems have been developed as a way to recreate human creativity. These systems implement models that aim to create stories similar to those written by humans. To date, some models have successfully imitated human creativity by creating interesting short stories about a particular domain. However, story-generation systems are far from creating stories of longer length and in multiple domains.

Earlier storytelling models focused on creating stories that simply made sense and followed the structure of short stories. These models followed problem-solving techniques where the main character logically performed a sequence of steps in order to achieve its goal. Later models focused on creating stories that are interesting and novel by using a wider range of techniques. The shift from problem-solving techniques to new techniques was necessary to ensure that stories maintained their uniqueness.

Computers as creative storytellers began in 1976 with the program TALE-SPIN (Meehan, 1976). This program showed that a computer could write stories and that the human creative process could be modeled up to some point using different techniques. The success of TALE-SPIN led to the development of several different story-generation systems that wrote short stories and implemented different models.

This section presents an overview of the most important story-generation systems: TALE-SPIN, MINSTREL, BRUTUS, MEXICA, and Virtual Storyteller. I evaluated the systems using the following criteria:

1. Author's reason for creating the system
2. System's contribution to the field of computers as storytellers
3. System's architecture

I then followed the evaluation by my personal critic of the system. This section concludes with a summary of the most important contributions each system brought to the field of computers as storytellers.

TALE-SPIN

Meehan (1976, p. 12) asked the question “What do you need in order to make up a good story?”. His answer was TALE-SPIN, a model that tried to recreate the way humans create stories. The model focused on the steps we humans take in order to achieve a certain goal. The model simulated how we solve problems with the available knowledge we possess. Thus the model is a problem-solving model.

TALE-SPIN was not the first story-generation system. However, it was the first to try to recreate the way we humans create stories. TALE-SPIN shed light to a wide range of techniques we use to perform this task, however, as is to be expected, future models improved greatly on the TALE-SPIN model. The creators of these models learned that stories should not just describe the sequence of actions a character performs in order to achieve a goal. A story is much more than that. This knowledge was TALE-SPIN greatest contribution to the field.

TALE-SPIN had three main components, which were a problem solver, an assertion mechanism, and an inference maker. The problem solver produced goals and events in the story given a specific goal. The assertion mechanism took an event and added it to the world model, which stored all the knowledge up to a certain point in the story. Finally, the inference maker produced a set of future events given a specific event. After the simulation was performed an English generator engine was called to generate the English sentences. It is important to note that each character along with the “narrator” itself had its specific world model, that is, its own set of knowledge and beliefs about the world.

TALE-SPIN required the user to specify characters, their personalities, and their relationships and would then use this information to create a story. Meehan considered these elements outside the

scope of simulating the way we humans create stories. The main character had a goal that needed to be accomplished. TALE-SPIN would use problem-solving techniques to develop the sequence of steps needed to achieve the goal and would then write the story using these steps.

Stories written by TALE-SPIN did not include many details. Stories were simply written as a sequence of logical steps, one leading to another. As Pérez y Pérez and Sharples (2004) noted, TALE-SPIN sometimes gave short and uninteresting stories because of its use of problem-solving techniques. The interestingness of the story depended heavily on the goal of the main character. For example, an interesting story was one where the main character wants to fool around with another character whereas an uninteresting story was one where the main character only rescues another character. Thus, TALE-SPIN did not succeed in creating a good short story, although Meehan did never claim he wanted TALE-SPIN to write good stories.

MINSTREL

Turner (1994) was interested in modeling the creative process we humans follow in order to create novel material. MINSTREL, his story-generation system, was a limited implementation of such a model applied to the field of computers as storytellers. Turner developed a set of criteria a storyteller should possess in order to create stories. These criteria are:

7. A storyteller must have an in-depth understanding of the stories he or she tells.
8. A storyteller must fashion his story to convey an interesting message.
9. A storyteller must be creative.
10. A storyteller must create stories that are aesthetically pleasing.

Turner noted that a human would have trouble following these criteria, and even more so a computer. However, by limiting the stories to a specific domain these criteria could be met. MINSTREL's domain is in the realm of stories about "King Arthur and the Knights of the Round Table."

Previous story-generation systems such as TALE-SPIN did not intend to model innovativeness. A story in those systems could be created twice if the right criteria was met. Creativity and innovation appeared in story-generation systems with MINSTREL. However, MINSTREL was not much different from previous models in that it still was a problem-solving model.

MINSTREL's architecture was developed to address Turner's storyteller criteria. Creativity is addressed by the use of TRAMs, which stand for Transform-Recall-Adapt Methods. TRAMs emulate case-based systems, which are systems that use prior information on how to solve problems to come up with new ways to solve a specific problem. The Transform process takes a specific problem and transforms it to a similar one. The Recall process then tries to find this new problem within the database. If it is in, then the Adapt process uses the specific solution that was used to solve this problem on the original problem. Therefore, with some changes to the solution of the transformed problem, the stories are ensured that they have creativity and novelty in them. Furthermore, to ensure that stories created by MINSTREL could not be repeated, they were stored in memory and used later to create new stories.

To address the other criteria MINSTREL uses four different author-level goals. Thematic goals ensure that a story will be interesting by having interesting themes. Consistency goals make sure that a story is consistent. Drama goals are concern with the artistic quality of a story. Finally, presentation goals are used to present the story in an appealing form.

MINSTREL stories were good, however, they were not at the same level as stories created by humans. Turner took a lot of things for granted when creating MINSTREL such as assuming a theme alone would suffice to make a story interesting. Some future models do not take this factor for granted and have techniques to measure interestingness.

BRUTUS

Bringsjord and Ferrucci (2000) had three reasons for developing the BRUTUS architecture. The first one was to prove whether we ourselves are machines. If we are, then our

creative process can be simulated by a machine. The second was to prove that logic can be used to create creativity in a computer. Finally, the third reason is, supposing they are able to create a human level writer BRUTUS version, for the money they would be able to make off it.

The BRUTUS architecture acknowledged the fact that in order for a story-generation system to write human level stories it will need a wide range of components. Each of these components will then be able to change a story in a specific way. The authors called this use of components architectural differentiation. Furthermore, the authors of the system came up with seven criteria, that they call the seven magic desiderata, which they believe a story-generation system should be able to meet in order to write human level stories. Criterion seven, present the story in an appealing literary prose, is accomplished by BRUTUS1, the first version of the BRUTUS architecture. Finally, BRUTUS1 did not implement MINSTREL's notion of creativity and novelty. Stories in BRUTUS1 could be repeated since there is nothing that impedes it.

The BRUTUS architecture consists of multiple components to ensure that stories vary among different dimensions. The authors divide the architecture into the knowledge level and process level. The knowledge level is divided into domain knowledge, linguistic knowledge, and literary knowledge. Domain knowledge holds knowledge about a specific domain objects. Linguistic knowledge holds knowledge necessary to produce natural language. Literary knowledge is knowledge about how to create a literary work. The process level is divided in thematic concept instantiation, plot generation, story structure expansion, and language generation. Thematic concept instantiation selects a theme, which has a detailed description of how the story should be in order to follow this theme. Plot generation develops the theme. Story structure expansion makes sure that the story is congruent. Finally, language generation produces a natural language story out of the story developed by BRUTUS.

Stories written by BRUTUS1 seem like they were written by a human author. However, the fact that BRUTUS1 can only handle one theme, betrayal, and only handles this theme inside a university setting makes stories, other than the first one a person reads, uninteresting. The authors

also assume that the theme and the wide range of components will make a story interesting, which may or may not be true. Future systems like MEXICA developed a way to measure how interesting a story is.

MEXICA

Pérez y Pérez had four main reasons for creating MEXICA (1999). The first one was to create a computer model of Sharples' account of writing, where writing is done in a process called engagement-reflection, to test whether Sharples' account explained how writers write. The second was to create a story-generation system that did not use problem-solving techniques or predefined story-structures, as earlier story-generation systems had done. The third was to create a system that produced novel and interesting stories. Finally, he wanted to create a system that users could experiment with using the different constraints involved in Sharples' account.

MEXICA did not write stories. It did, however, create frameworks for stories. MEXICA brought two important contributions to the field of computers as storytellers: it did not follow the conventional approach of creating stories using problem-solving techniques, and it implemented a method that quantifies how interesting a story is.

The creation of a story in MEXICA began when the user defined possible story-actions and some previous stories within the previous stories database, which holds MEXICA body of stories. MEXICA then generated stories using the engagement-reflection cognitive account of writing. The engagement process was executed first. During engagement MEXICA searched its previous stories database for stories that had similar story-world contexts (discussed shortly) and retrieved them. MEXICA then used the stories to generate new material by changing some aspects of the stories and applying them to the current story. The reflection process was then executed. During reflection MEXICA verified the coherence of the story, evaluated its novelty and interest, and break impasses. The coherence of the story was verified by making sure that all the necessary preconditions to actions in the story were present. If they were not, MEXICA added them to the

story along with their necessary preconditions. MEXICA used the concept of tension to evaluate if a story was interesting. Tension was recorded as a numerical value, which was retrieved by evaluating certain actions that had a fixed amount of tension, such as killing. The novelty of the story was assured by reviewing other stories in the previous stories database and making sure that the current story was different from them. When the system encountered an impasse, something that forbade the story from moving along, MEXICA searched the previous stories database for possible ways to break the impasse.

MEXICA created a story-world context for each of the characters where post-conditions were stored. The story-world context of a character was everything that the character had experienced during the story. Post-conditions were experiences not written directly in the story, but that had an effect on the story-world context of a character. That is, if, for example, one person tried to kill another one in the story, then the post-condition of hate between one character and another was stored. Furthermore, only the story-world contexts of the characters present at the given time and place when an action happened were changed.

MEXICA's story frameworks were detailed and interesting. Unfortunately, those frameworks were not translated to natural language, so it is difficult to evaluate the system. However, the system brought interesting ideas to the field of computers as storytellers that should be implemented in models to come.

VIRTUAL STORYTELLER

The Virtual Storyteller was created as part of the AVEIRO project at the University of Twente (Faas, 2002). The AVEIRO project aims to create a virtual environment filled with autonomous embodied agents. The Virtual Storyteller is part of the Virtual Music Centre inside the virtual environment and will be used to entertain people visiting the Centre.

The Virtual Storyteller system used agents to develop a story, much like other story-generation systems, i.e., TALE-SPIN. The approach to creating stories is novel in that it introduces the notion of a director, a narrator and a presenter.

The system uses agents, which have a knowledge base, to create, direct, write, or present the story. Agents are divided in four categories: characters, director, narrator, and presenter. The characters have goals that they try to achieve in the plot. The director tries to make the characters create a good plot. It does this by introducing characters, giving them goals, and/or restricting characters actions. Once the plot is created by the characters and director, the plot is sent to the narrator that translates it to written text. The presenter then presents it to an audience.

At the time of this writing there is no story created by the system using more than one agent, so it is difficult to criticize the system. However, their system seems to be promising and I expect it to create good stories.

Summary

Story-generation systems are far from creating novel and interesting stories across multiple domains. TALE-SPIN's approach was interesting, but one who has read a story authored by the system would find the stories themselves uninteresting. MINSTREL's stories, though, followed a problem-solving approach similar to that used by TALE-SPIN, but by implementing other techniques stories in MINSTREL were more interesting than in TALE-SPIN. The BRUTUS system was designed as an architecture that allows future improvement. Stories created by BRUTUS are interesting and well written, but get boring after reading a few because they all have the same theme. Future implementations will most likely allow for different themes. MEXICA story frameworks are interesting and its approach to creating them is novel. Work in MEXICA is still being done. Finally, the Virtual Storyteller approach of using intelligent agents to generate a story seems promising. Future story-generation systems should learn from all these

models and try to implement the most useful things each system brought to the field of computers as storytellers.

I developed LUKIS TALES using agents because I wanted to represent different characters each with his or her own goals. LUKIS TALES is a system where agents can roam free and create conversations with whomever they want based on their needs. Unlike the story-generation systems discussed in this chapter, LUKIS TALES's goal is to output interesting conversations created by agents and not an interesting narrative.

CHAPTER 3

AGENT CONVERSATION SYSTEMS

Researchers implementing agent-agent conversational systems study the elements that are involved in a conversation and how to represent them. A simulation of conversation must have a means through which it allows the participants to communicate and it may implement a mutual model of conversation. In addition, it may be implemented in such a way that it allows human conversants to participate with it, thus becoming an interactive system.

Speech-act theory

Speech-act theory is used as a way to allow the agents in a conversational system to communicate and reason. Speech-act theory was developed by Austin (as cited in Novick, 1988; Traum, 1999) as a way to understand the functions of language. He saw language as a way to change the state of the world and not just transmit information. These speech-acts could represent the way humans communicate in order to reach a goal. He distinguished three different speech acts:

11. Locutionary act- act of saying something
12. Illocutionary act- act performed in saying something
13. Perlocutionary act- act performed by saying something

Locutionary acts are then the utterances that the speakers generate. For example “the store closes in five minutes.” Illocutionary acts are what the speaker tries to convey to the hearer when he or she generates the utterance. For example, the utterance used in the previous example tries to convey to the hearer that the store is about to close. Perlocutionary act is the effect the speaker intends to cause on the listener once the utterance is heard. For example, from the previous examples, the speaker tries to make the hearer finish shopping fast or else he or she might not be able to buy anything because the store is going to close.

Mutual model of conversation

Participants in a conversation must maintain a mutual model of the conversation in order for it to be useful. If the participants in a conversation have different views of the conversations then their messages will be interpreted incorrectly and the conversation will not lead any of its participants to their desired goals.

Beliefs about other participants is one way to help maintain a conversation consistent. Novick and Ward (1993) developed a multi-agent system that represented and reasoned about dialogue in terms of the mutual beliefs of agents in a conversation. In their system, each agent had a belief structure where it stored its beliefs. Each belief was represented in the following form:

belief(proposition, truth-value, belief-group).

Proposition was the item about which a belief was held. Truth-value represented what the agent thought the other agents believed about the item. Belief-group represented the group that the agent thought held the specific belief.

With this model, Novick and Ward were able to show how beliefs could be represented in a computer model to help maintain a mutual model of the conversation.

Collaborative storytelling systems

Ryokai et al. (2003) developed an interactive conversational system where an embodied conversational agent, Sam, interacted with children in order to improve their literacy skills. Sam interacted with the children by telling them stories and then allowing the children to tell stories back to Sam. Sam used decontextualized language, which is language not bound to spatial or historical context, in its stories. The children, upon hearing Sam's stories, then began to use decontextualized language, which allowed them to become better storytellers.

Conclusion

Agent-agent conversational systems must account for several elements found in human conversations. They must be able to represent messages from one participant to another and they may implement a mutual model of conversation. Story-generation systems may be implemented in such a way that they create stories using conversations. For my system, LUKIS TALES, I represented messages between agents using speech-acts. I did not, however, implement a mutual model of conversation nor did I make my system interactive.

CHAPTER 4

LUKIS TALES

LUKIS TALES is an implementation of a story-generation system where agents communicate with each other to construct conversations. This chapter discusses how agents can be created to appear as characters from a story, the approach used to create the agents, and how they are able to send messages to each other.

Agents as characters

Russell and Norvig defined intelligent agents as “anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.” (2003, p. 32). Agents as characters started with TALE-SPIN, which allowed the agents to do whatever they needed to do to achieve their goals. The system would then use the characters' actions to write the story. The advantage of using agents, rather than a single narrator to write the story, consists of having characters with richer personalities, which make stories more interesting. Agent architectures are constructed depending on the goals the agent is supposed to fulfill. I considered two agent architectures for LUKIS TALES. The first one, advanced synthetic characters, was developed by Bringsjord et al. (2005) and is concerned with having characters that not only appear to have human characteristics but actually have them. Bringsjord defined advanced synthetic characters as:

- x is a person if and only if x has the capacity
14. to “will,” to make choices and decisions, set plans and projects — autonomously;
 15. for consciousness, for experiencing pain and sorrow and happiness, and a thousand other emotions — love, passion, gratitude, and so on;
 16. for self-consciousness, for being aware of his/her states of mind, inclinations, preferences, etc., and for grasping the concept of him/herself;

17. to communicate through a language;
 18. to know things and believe things, and to believe things about what others believe, and to believe things about what others believe about one's beliefs (and so on);
 19. to desire not only particular objects and events, but also changes in his or her character;
 20. to reason (for example, in the fashion exhibited in the writing and reading of this very paper).
- (Bringsjord et al., 2005, pp. 1-2)

Although interesting, this architecture is beyond the scope of a story-generation system. The only goal a story-generation system has is to create interesting and novel stories. Agents that appear human, instead of *being*, should be enough. So, an architecture that focuses on creating believable emotional agents should be sufficient. The architecture, developed by Reilly for the Oz project, has as its goal to create interactive believable agents (1996). He defined believable characters as “characters that seem to be alive and that an audience has emotions for or about.” (p. 10). Furthermore, he listed the characteristics of believable characters in the arts:

1. Believable agents may not be intelligent.
2. Believable agents may not be realistic.
3. Believable agents will have strong personalities. (pp. 10-12)

To develop agents that fit these characteristics he, along with members of the Oz project, opted for a broad agent architecture. Broad agent architectures have three important properties:

4. They have a broad set of capabilities.
5. Each capability is typically (but not necessarily) somewhat shallow.
6. All of the capabilities are tightly integrated. (p. 12)

Another advantage to using believable emotional agents is that they are designed to deal with emotions and relationships.

LUKIS TALES uses agents to generate conversations. The agents were created with Reilly's three characteristics of believable characters in mind. The agents follow a script when establishing a conversation, thus they are not intelligent. They are not realistic because they do not model

human cognition as used in conversations. And they have strong personalities in that they act as their human counterpart might. Furthermore, the current implementation of the agents only captures two of the three important properties a broad agent architecture should have. Each capability an agent has is shallow and the capabilities are tightly integrated. Unfortunately, their capabilities are limited.

Agent architecture

The agents were developed using a symbolic cognitive architecture named Soar (Laird & Congdon, 2006). Soar is similar to logic programming languages in that it uses rules, or productions as are named in Soar, to develop agents. I used Soar because I found it intuitively easy to use because of the way the rules are written, which I will explain later. Agents can be developed separately from each other and then made to interact with each other using the SML, which stands for Soar markup language, and another programming language such as Java or C++. Rules in Soar start with a state and follow with any number of conditions that test the working memory elements (WME), called facts in other rules engines, of the current state. The representation of WME that I created can be seen in Figure 1 below. The S in the center of the figure indicates the state of the agent. The edges coming out of it indicate attributes of the state. One of the attributes of a state is agent, which represents an agent. An agent can have a particular emotion, goals, an environment, and so on. I decided to have the agent structure be different than the overall state structure because I figured I would also need to represent other things meta to the agent, such as the different environments in the world the agents lives in. Appendix B shows some selected productions.

I had two major problems when implementing my system. The first problem was my lack of knowledge of Soar's syntax. I spent countless hours researching how to write rules that worked as I intended them to work. Once I figured how to write rules correctly writing more was relatively easy. The second problem was creating a representation of the WME that allowed me to

The agents are able to send messages to each other by means of a program written in C++ that uses the productions written in Soar and the Soar markup language. The agents communicate by sending speech-acts and arguments to each other. These speech acts and their arguments are then interpreted by the other agent who reasons about them and sends a speech act and arguments as a reply. In my implementation of the agents, the attribute named performative represents the speech acts. The attributes argument1, argument2, and argument3 are used if the speech act requires arguments. Furthermore, the attribute content is a natural language representation of what the agents are saying to each other.

Summary

Software agents can be made to represent characters of a story by providing them with the kind of reasoning their character counterpart might have. To develop these agents a number of programming languages can be used. I decided to use Soar because it was more intuitive. After developing the agents I used the Soar markup language combined with a C++ application to make them communicate with each other.

CHAPTER 5

EVALUATION OF LUKIS TALES IN TERMS OF ELEMENTS OF LITERARY WORKS

Conversations can be considered literary works as long as they contain literary elements such as plots, drama, and incidents. This section discusses several literary elements that are used to evaluate LUKIS TALES in terms of the conversations it creates. These elements include drama, Freytag's Pyramid, plot, characters, and incidents.

Drama

The word drama comes from the Greek word for action. According to Field (2005, p. 41), regarded as the most sought-after screenwriting teacher in the world by Hollywood Reporter, “all drama is conflict.” Characters have dramatic needs, which are their goals. These needs, along with the obstacles that get in the way of accomplishing the needs, move a story forward. Stories have dramatic premises, which are “what a story is about” (2005, p. 3), and dramatic situations, “circumstances surrounding the action” (2005, p. 3). Having goals, conflicts, and resolving those goals and conflicts is what drama is all about.

LUKIS TALES can only generate a limited series of conversations. They are presented in a random manner and they consider the current state of the agent. The content of the conversations, however, is scripted and as so will not change. Evaluating LUKIS TALES in terms of drama means evaluating the conversations scripted inside the system. Conversation sample 1, which can be found in appendix A, is an example of a dramatic conversation. In this conversation an agent that represents a graduate student asks another agent, that represents an advisor, whether his work is good or not. The advisor then deliberates on whether he should let the student know that his work is good or not. The advisor then decides to tell the student that it is not because he wants to keep the graduate student longer. The conflicting goals of both agents then create drama.

Freytag's Pyramid

Freytag (1863) divided drama into five parts, which can be represented as a pyramid. The parts are exposition, rising action, climax, falling action, and catastrophe. Exposition introduces the background of the story. It presents the theme, setting, characters and gives hints about the conflict in the story. During rising action the conflict is introduced and grows by having the characters go through obstacles. The climax is when the tension is at its highest point. During the climax the story makes a turn for better or worse. During falling action the conflict with the antagonist is resolved. Finally, during catastrophe the audience learns how the hero is worse than how he or she was before.

LUKIS TALES is not able to generate conversations that follow Freytag's Pyramid. Every conversation generated by the system starts with exposition, as can be seen in sample conversation 2, where an agent, the main protagonist, explains his or her dilemma. Rising action may occur, but is not guaranteed due to the random nature of the conversation. If, for example, the main protagonist were to encounter his friend and his enemy then rising action would occur. However, if the agent encounters his or her advisor first, then no rising action occurs. Climax always occurs when the advisor and the graduate student start a conversation with each other. Falling action and catastrophe never occur because no conversation currently implemented in the system may fall into either of those two categories.

Plot

The plot is a sequence of events, which involves the characters in conflict. The plot immerses the audience into the characters' lives. For a plot to be interesting it must leave questions and expectations, as well as have characters that are interesting. Plot points are defined as "any incident, episode, or event that hooks into the action and spins it around in another direction" (Field, 2005, p. 26). Stories can be divided into beginning, middle, and end, where plot points are used to change between one part to the next.

The recurring plot in LUKIS TALES is whether the protagonist will graduate or not. The agents inside LUKIS TALES succeed in representing what they are supposed to represent. Conversation sample 3 and 4 show representations of the friend and enemy of the protagonist respectively. In conversation 3 the friend succeeds in his role by supporting the protagonist and providing emotional comfort to him or her. In conversation sample 4 the enemy personifies his or her role by telling the protagonist that his work is bad. LUKIS TALES succeeds in these criteria for plot. LUKIS TALES does not succeed in the criteria that a story must be clearly divided in beginning, middle and end. LUKIS TALES has a clearly defined beginning and middle, but it is unclear where exactly the story ends.

Creating Characters

According to Field (2005), good characters have four essential qualities:

21. The characters have a strong and defined dramatic need.
22. They have an individual point of view.
23. They personify an attitude.
24. They go through some kind of change, or transformation.

Dramatic needs are the character's goals during the story. Point of view is how the characters see their world. Attitude reveals a character's personal opinion by having the character act a certain way.

Having all these qualities in the characters will most likely guarantee interesting characters in a story, which along with other literary elements will contribute to creating an interesting story.

In addition to these qualities, characters must also possess life profiles to make them believable.

Field distinguished three life profiles each character should have:

4. Professional life, which deals with everything about the character's professional life such as where the character works and his/her relationships with coworkers.

5. Personal life, which deals with everything about the character's personal life such as his/her love relationships and family relationships.
6. Private life, which deals with everything about the character's private life such as what the character does and thinks when he/she is alone.

LUKIS TALES characters all satisfy the first three criteria proposed by Field. They have a dramatic need, which goes from graduating to helping a friend. They have an individual point of view, which goes from liking someone to disliking someone else. And they all personify an attitude, which goes from being friendly to being mean. No character satisfies criterion four in the sense Field explains it as. All the characters inner self and world remain the same during the conversations. In this current implementation of LUKIS TALES none of the agents support any of the three life profiles.

Incident

Incidents are events that occur in relationship to something else. Characters need to go through incidents in order to be revealed dramatically. Field (2005) noticed two important incidents during a screenplay:

7. Inciting incident, which sets the story in motion.
8. Key incident, which is what the story is about.

“The inciting incident always leads us to the key incident,” Field (2005, p. 133) continued. The key incident is especially important because it will be recalled during the story. If the key incident is not interesting, then chances are the story will also not be interesting.

LUKIS TALES combines the inciting incident and the key incident in an initial soliloquy by the protagonist as can be seen in conversation sample 2. The soliloquy sets the story in motion by introducing the protagonist and tells us what the story is about by introducing us to his or her goal.

Summary

To create a good literary work, several literary elements must be used. A structure must be followed; otherwise the story will not be understood or could face the risk of being boring. The story must also have an interesting incident, which will grab the audience and keep them interested. Characters must also be interesting and believable. Having all these elements may not be enough to ensure a great literary work, but it will, however, lead to a well structured and interesting one. The current implementation of LUKIS TALES is able to satisfy several of these literary elements.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Conclusion

LUKIS TALES suggests that agent-agent simulation of conversation can be extended to create literary works. It does so by proposing criteria of what constitutes a literary work and then succeeding in satisfying this criteria. LUKIS TALES is also the first story generation system to be written in Soar. Future story-generation enthusiasts may consider developing their agents using Soar because LUKIS TALES was moderately able to show that developing agents in Soar is easier than in other logic programming languages. Furthermore, using the Soar markup language it is also easy to make the agents interact with one another.

LUKIS TALES fails, not unlike other systems, to create a wide range of stories. The conversations the agents generate are scripted and do not make much use of a particular domain of knowledge. Because they are scripted, LUKIS TALES is not creative. Its architecture also fails to account for several elements commonly found in conversations such as body language, turn-taking, and correction. Furthermore, LUKIS TALES does not implement a shared model of conversation or beliefs about other agents.

Future work

Story generation systems are still far from imitating the quality and breadth of stories created by a human writer. To close the gap, the next step is to create a system that can reason about a vast amount of knowledge and can use this knowledge to create stories that span multiple domains. The goal of the Open Mind Common Sense project, developed by Singh (2002), is to create a large common-sense knowledge base and ways to reason about it. By making use of projects such as this one, story-generation systems may be able to create richer stories.

Future work in simulation of conversations to create literary works should also account for a shared model of conversation between the participants, body language that may be used by the participants, and a theory of mind (Boden, 2004).

REFERENCES

- Boden, M. A. (2004). *The creative mind: myths and mechanisms*. New York: Routledge.
- Bringsjord, S., & Ferrucci, D. A. (2000). *Artificial Intelligence and Literary Creativity: Inside the Mind of Brutus, A Storytelling Machine*. New Jersey: Lawrence Erlbaum Associates.
- Bringsjord, S., Khemlani, S., Arkoudas, K., McEvoy, C., Destefano, M., Daigle, M. (2005). *Advanced Synthetic Characters, Evil, and E*. Game-On 2005, 6th International Conference on Intelligent Games and Simulation, (Ghent-Zwijnaarde, Belgium: European Simulation Society), pp. 31–39.
- Faas, S. (2002). *Virtual Storyteller: An approach to computational story telling*. Masters thesis, University of Twente.
- Field, S. (2005). *Screenplay: The Foundations of Screenwriting*. New York: Delta.
- Freytag, G. (2004). *Technique of the Drama: An Exposition of Dramatic Composition and Art*. Hawaii: University Press of the Pacific.
- Laid, J. E., Congdon, C. B. (2006). *The Soar User's Manual Version 8.6.3*. Retrieved December 15, 2006, from <http://ai.eecs.umich.edu/soar/sitemaker/docs/manuals/Soar8Manual.pdf>
- Meehan, J. (1976). *The metanovel: Writing stories by computer* (Tech. Rep. #74, doctoral dissertation). Yale University, Dept. of Computer Science.
- Novick, D. (1988). Control of Mixed-Initiative Discourse Through Meta-Locutionary Acts: A Computational Model, Technical Report CIS-TR-88-18, Department of Computer and Information Science, University of Oregon.
- Novick, D., and Ward, K., (1993). Mutual beliefs of multiple conversants: A computational model of collaboration in air traffic control, Proc. of AAAI'93, Washington, DC, July, 1993, 196-201.
- Pérez y Pérez, R., & Sharples, M. (2004). *Three computer-based models of storytelling: BRUTUS, MINSTREL and MEXICA*. Knowledge-Based Systems, 17, 15-29.
- Pérez y Pérez, R. (1999). *MEXICA: a computer model of creativity in writing*. PhD dissertation, University of Sussex.
- Traum, D. R. (1999). Speech Acts for Dialogue Agents. In M. Wooldridge and A. Rao, editors, *Foundations And Theories Of Rational Agents* (pp. 169-172). Kluwer Academic Publishers.
- Turner, S. R. (1994). *The Creative Process: A Computer Model of Storytelling and Creativity*. New Jersey: Lawrence Erlbaum Associates.
- Reilly, W. S. N. (1996). *Believable Social and Emotional Agents*. PhD dissertation, Carnegie Mellon University.

Ryokai, K., Vaucelle, C., Cassell, J. (2003). *Virtual Peers as Partners in Storytelling and Literacy Learning*. *Journal of Computer Assisted Learning* 19(2), pp. 195-208.

Russell, S. J., Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. New Jersey: Prentice Hall.

Singh, Push. (2002). The Open Mind Common Sense Project. Available:
<http://www.kurzweilai.net/meme/frame.html?main=/articles/art0371.html>

APPENDIX A – Sample conversations

LUKIS TALES Simulation in Soar	Interpretation in English
<p>===Agent talk=== ag1 says:</p> <p>0. (graduate-student ^envi street) (graduate-student ^performative show) (graduate-student ^content I am presenting my work to my advisor.) (graduate-student ^argument1 work) (graduate-student ^receiver advisor)</p>	<p>- Graduate student shows work to advisor.</p>
<p>===Agent talk=== ag4 says:</p> <p>0. (advisor ^envi street) (advisor ^performative evaluate) (advisor ^content My graduate student's work is good.) (advisor ^argument1 work-good) (advisor ^receiver me)</p>	<p>- Advisor soliloquizes: My graduate student's work is good.</p>
<p>1. (advisor ^envi street) (advisor ^performative inform) (advisor ^content I am thinking.) (advisor ^argument1 thinking) (advisor ^receiver graduate-student)</p>	<p>- Advisor says to graduate student: I am thinking.</p>
<p>===Agent talk=== ag4 says:</p> <p>0. (advisor ^envi street) (advisor ^performative keep) (advisor ^content I must keep my</p>	<p>- Advisor soliloquizes: I must keep my graduate student.</p>

<p>graduate student.)</p> <p>(advisor ^argument1 graduate-student)</p> <p>(advisor ^receiver me)</p> <p>1. (advisor ^envi street)</p> <p>(advisor ^performative answer)</p> <p>(advisor ^content Not good enough.)</p> <p>(advisor ^argument1 good-enough)</p> <p>(advisor ^argument2 no)</p> <p>(advisor ^receiver graduate-student)</p>	<p>- Advisor says to graduate student: Not good enough.</p>
---	---

Conversation sample 1. Conversation between a graduate student and his or her advisor.

<p>LUKIS TALES Simulation in Soar</p> <p>====Agent talk==== ag1 says:</p> <p>0. (graduate-student ^envi apartment)</p> <p>(graduate-student ^performative consider)</p> <p>(graduate-student ^content I am considering getting my advisor's approval.)</p> <p>(graduate-student ^argument1 get)</p> <p>(graduate-student ^argument2 advisor)</p> <p>(graduate-student ^argument3 approval)</p> <p>(graduate-student ^receiver me)</p> <p>====Agent talk==== ag1 says:</p>	<p>Interpretation in English</p> <p>- Graduate student soliloquizes: I should get my advisor's approval.</p>
---	--

<p>0. (graduate-student ^envi apartment) (graduate-student ^performative do) (graduate-student ^content I considered the quality of my work bad, so I had to work on it before going after my advisor's approval) (graduate-student ^argument1 improve) (graduate-student ^argument2 work) (graduate-student ^receiver me) ===Agent talk=== ag1 says:</p>	<p>- Graduate student soliloquizes: I have to improve my work.</p>
<p>0. (graduate-student ^envi apartment) (graduate-student ^performative get) (graduate-student ^content My work is good. Now it is time to get my advisor's approval. My emotion is good!) (graduate-student ^argument1 approval) (graduate-student ^argument2 advisor) (graduate-student ^receiver me)</p>	<p>- Graduate student soliloquizes: I am ready to get my advisor's approval.</p>

Conversation sample 2. Soliloquy of graduate student.

LUKIS TALES Simulation in Soar	Interpretation in English
<p>===Agent talk=== ag1 says:</p> <p>0. (graduate-student ^envi apartment) (graduate-student ^performative ask) (graduate-student ^content What do you think of the quality of my work?) (graduate-student ^argument1 work) (graduate-student ^argument2 quality) (graduate-student ^receiver friend)</p> <p>===Agent talk=== ag2 says:</p> <p>0. (friend ^envi apartment) (friend ^performative answer) (friend ^content The quality of your work is good.) (friend ^argument1 work) (friend ^argument2 quality) (friend ^argument3 good) (friend ^receiver graduate-student)</p> <p>===Agent talk=== ag1 says:</p>	<p>Interpretation in English</p> <p>- Graduate student asks friend: What do you think of the quality of my work?</p> <p>- Friend replies to graduate student: The quality of your work is good.</p>

<p>0. (graduate-student ^envi apartment) (graduate-student ^performative ask) (graduate-student ^content Do you think it is good enough for my advisor?) (graduate-student ^argument1 good-enough-for) (graduate-student ^argument2 advisor) (graduate-student ^receiver friend) ===Agent talk=== ag2 says:</p>	<p>- Graduate student asks friend: Do you think it is good enough for my advisor.</p>
<p>0. (friend ^envi apartment) (friend ^performative answer) (friend ^content I believe it is.) (friend ^argument1 good-enough-for) (friend ^argument2 advisor) (friend ^argument3 yes) (friend ^receiver graduate-student)</p>	<p>- Friend replies to graduate student: I believe it is good enough for your advisor</p>

Conversation sample 3. Conversation between a graduate student and his or her friend.

LUKIS TALES Simulation in Soar	Interpretation in English
<p>===Agent talk=== ag3 says:</p>	
<p>0. (enemy ^envi street)</p>	<p>- Enemy says to graduate student: Your</p>

<p>(enemy ^performative tell)</p> <p>(enemy ^content Your work is bad.)</p> <p>(enemy ^argument1 quality-work)</p> <p>(enemy ^argument2 bad)</p> <p>(enemy ^receiver graduate-student)</p> <p>===Agent talk=== ag1 says:</p> <p>0. (graduate-student ^envi lab)</p> <p>(graduate-student ^performative inform)</p> <p>(graduate-student ^content I do not care.)</p> <p>(graduate-student ^argument1 care)</p> <p>(graduate-student ^argument2 no)</p> <p>(graduate-student ^receiver enemy)</p> <p>1. (graduate-student ^envi lab)</p> <p>(graduate-student ^performative end-discourse)</p> <p>(graduate-student ^content I decided to end conversation.)</p> <p>(graduate-student ^receiver enemy)</p>	<p>work is bad.</p> <p>- Graduate student replies: I do not care.</p> <p>- Graduate student decides to end the conversation.</p>
---	--

Conversation sample 4. Conversation between a graduate student and his or her enemy.

APPENDIX B – Selected productions written in Soar

```
# If agent has not been initialized
sp {agent*propose*initAgent
  (state <s> ^io.input-link.init <init>
    -^agent)
```

```

-->
  (<s> ^operator <o> +=)
  (<o> ^name initAgent)
}

# Initialize the agent with the information that the
# conversation manager, Togashi, sends me
sp {agent*apply*initAgent
  (state <s> ^operator.name initAgent
    ^io.input-link <io>)
  (<io> ^init <init>)
  (<init> ^agent <in-agent>)
  (<in-agent> ^emotion <state>
    ^envi <ag-envi>
    ^content <content>
    ^performative <performative>
    ^name <name>
    ^title <title>
    ^work <q-work>)
-->
  (<s> ^agent <agent>)
  (<agent> ^emotion <state>
    ^envi <ag-envi>
    ^goal <goal>
    ^name <name>
    ^title <title>
    ^work <work>)
  (<goal> ^content <content>
    ^performative <performative>
    ^receiver me
    ^sender none)
  (<work> ^quality <q-work>)
}

# If environment has not been initialized
sp {agent*propose*initEnvi
  (state <s> ^io.input-link.init <init>
    -^envi)
-->
  (<s> ^operator <o> +=)
  (<o> ^name initEnvi)
}

# Initialize the environment with the information that the
# conversation manager, Togashi, sends me
sp {agent*apply*initEnvi

```

```

(state <s> ^operator.name initEnvi
  ^io.input-link <io>)
(<io> ^init <init>)
(<init> ^envi <in-envi>)
(<in-envi> ^name <envi-name>)
-->
(<s> ^envi <envi>)
(<envi> ^name <envi-name>)
}

```

Selected productions 1. Initialization of the agents.

```

# If I have a goal, which is to graduate
sp {agent*propose*graduate
  (state <s> ^agent <agent>)
  (<agent> ^goal <goal>)
  (<goal> ^performative graduate
    -^status)
-->
(<s> ^operator <o> + =)
(<o> ^name graduate)
}

# Consider graduating
sp {agent*apply*graduate
  (state <s> ^operator.name graduate
    ^agent <agent>)
  (<agent> ^emotion <emotion>
    ^goal <goal>)
  (<goal> ^performative graduate
    -^status)
-->
(<agent> ^emotion bad
  ^goal <ag-goal>)
(<ag-goal> ^argument1 get
  ^argument2 advisor
  ^argument3 approval
  ^content |I am considering getting my advisor's approval.|
  ^out ok
  ^performative consider
  ^receiver me)
(<goal> ^status progress)
(<agent> ^emotion <emotion> -)
}

sp {agent*propose*work-quality-bad
  (state <s> ^agent <agent>)

```

```

(<agent> ^goal <goal>
  ^work <work>)
(<goal> ^argument1 get
  ^argument2 advisor
  ^argument3 approval
  ^performative consider
  -^status)
(<work> ^quality bad)
-->
(<s> ^operator <o> +=)
(<o> ^name work-quality-bad)
}

sp {agent*apply*work-quality-bad
  (state <s> ^operator.name work-quality-bad
    ^agent <agent>)
  (<agent> ^goal <ag-goal>
    ^work <work>)
  (<ag-goal> ^argument1 get
    ^argument2 advisor
    ^argument3 approval
    ^performative consider
    -^status)
  (<work> ^quality bad)
-->
  (<work> ^quality bad -)
  (<work> ^quality good)
  (<agent> ^goal <goal>)
  (<goal> ^argument1 improve
    ^argument2 work
    ^content |I considered the quality of my work bad, so I had to work on it before
going after my advisor's approval|
    ^out ok
    ^performative do
    ^receiver me)
  }

# If my work is good and I consider getting my advisor's approval, then
# propose to do so
sp {agent*propose*work-quality-good
  (state <s> ^agent <agent>)
  (<agent> ^goal <goal>
    ^work <work>)
  (<goal> ^argument1 get
    ^argument2 advisor
    ^argument3 approval

```

```

    ^performative consider
    -^status)
  (<work> ^quality good)
-->
  (<s> ^operator <o> + =)
  (<o> ^name work-quality-good)
}

# Get my advisors approval if my work is good
sp {agent*apply*work-quality-good
  (state <s> ^operator.name work-quality-good
    ^agent <agent>)
  (<agent> ^emotion <emotion>
    ^goal <goal>
    ^work <work>)
  (<goal> ^argument1 get
    ^argument2 advisor
    ^argument3 approval
    ^performative consider
    -^status)
  (<work> ^quality good)
-->
  (<goal> ^status complete)
  (<agent> ^goal <ag-goal>
    ^emotion good)
  (<ag-goal> ^argument1 approval
    ^argument2 advisor
    ^content |My work is good. Now it is time to get my advisor's approval. My
emotion is good!!|
    ^out ok
    ^performative get
    ^receiver me)
  (<agent> ^emotion <emotion> -)
}

```

Selected productions 2. Soliloquy of graduate student.

```

# If there is a valid message for me with no arguments
# Valid means both agents are in same environment
# and are involved in a discourse with each other
sp {agent*propose*in-valid-message-me
  (state <s> ^agent <agent>
    ^io.input-link.agent <in-ag>)
  (<in-ag> ^message <message>)
  (<message> -^argument1
    -^argument2

```

```

    -^argument3
    ^performative { <performative> <> dis-status-invite <> no-action }
    ^receiver <title>
    ^sender <in-sender>
    -^status)
  (<agent> ^title <title>
    ^discourse <dis>)
  (<dis> ^dis-status engage
    ^participant <in-sender>)
-->
  (<s> ^operator <o> + =)
  (<o> ^name in-invalid-message-me
    ^performative <performative>
    ^sender <in-sender>
    ^message <message>)
}

# Get new valid message for me
sp {agent*apply*in-invalid-message-me
  (state <s> ^operator <operator>
    ^agent <agent>)
  (<operator> ^name in-invalid-message-me
    ^performative <performative>
    ^sender <sender>
    ^message <message>)
-->
  (<message> ^status complete)
  (<agent> ^goal <ag-goal>)
  (<ag-goal> ^performative <performative>
    ^sender <sender>)
}

#####

# If there is a valid message for me with one arguments
# Valid means both agents are in same environment
# and are involved in a discourse with each other
sp {agent*propose*in-invalid-message-me-1argument1
  (state <s> ^agent <agent>
    ^io.input-link.agent <in-ag>)
  (<in-ag> ^message <message>)
  (<message> ^argument1 <argument1>
    -^argument2
    -^argument3
    ^performative { <performative> <> dis-status-invite}
    ^receiver <title>

```

```

    ^sender <in-sender>
    -^status)
-{{(<message> ^argument1 move
    ^performative inform)}}
(<agent> ^title <title>
    ^discourse <dis>)
(<dis> ^dis-status engage
    ^participant <in-sender>)
-->
(<s> ^operator <o> + =)
(<o> ^name in-valid-message-me-1argument1
    ^argument1 <argument1>
    ^performative <performative>
    ^sender <in-sender>
    ^message <message>)
}

# Get new valid message for me
sp {agent*apply*in-valid-message-me-1argument1
    (state <s> ^operator <operator>
        ^agent <agent>)
    (<operator> ^name in-valid-message-me-1argument1
        ^argument1 <argument1>
        ^performative <performative>
        ^sender <sender>
        ^message <message>)
-->
(<message> ^status complete)
(<agent> ^goal <ag-goal>)
(<ag-goal> ^argument1 <argument1>
    ^performative <performative>
    ^sender <sender>)
}

```

Selected productions 3. Receiving input from other agents.

```

#graduate
#####
sp {agent*propose*ask-quality-work
    (state <s> ^agent <agent>)
    (<agent> ^discourse <dis>
        ^title graduate-student)
    (<dis> ^dis-status engage
        ^participant <title> << friend >>)
-{{(<agent> ^goal <goal>)
    (<goal> ^argument1 work
        ^argument2 quality

```

```

    ^performative ask
    ^receiver <title>)}
-->
(<s> ^operator <o> + =)
(<o> ^name ask-quality-work
 ^participant <title>)
}

sp {agent*apply*ask-quality-work
 (state <s> ^operator <o>
 ^agent <agent>)
 (<o> ^name ask-quality-work
 ^participant <participant>)
-->
(<agent> ^goal <goal>)
(<goal> ^argument1 work
 ^argument2 quality
 ^content |What do you think of the quality of my work?|
 ^out ok
 ^performative ask
 ^receiver <participant>)
}

sp {agent*propose*ask-good-enough-for-advisor
 (state <s> ^agent <agent>)
 (<agent> ^discourse <dis>
 ^goal <goal>)
 (<dis> ^dis-status engage
 ^participant <title>)
 (<goal> ^argument1 work
 ^argument2 quality
 ^argument3 good
 ^performative answer
 ^sender <title>
 -^status)
-->
(<s> ^operator <o> + =)
(<o> ^name ask-good-enough-for-advisor
 ^participant <title>)
}

sp {agent*apply*ask-good-enough-for-advisor
 (state <s> ^operator <o>
 ^agent <agent>)
 (<o> ^name ask-good-enough-for-advisor
 ^participant <participant>)
}

```

```

(<agent> ^goal <ag-goal>)
(<ag-goal> ^argument1 work
  ^argument2 quality
  ^argument3 good
  ^performative answer
  ^sender <title>)
-->
(<agent> ^goal <goal>)
(<goal> ^argument1 good-enough-for
  ^argument2 advisor
  ^content |Do you think it is good enough for my advisor?!
  ^out ok
  ^performative ask
  ^receiver <participant>)
(<ag-goal> ^status complete)
}

sp {agent*propose*answer-good-enough-for-advisor-yes
  (state <s> ^agent <agent>)
  (<agent> ^discourse <dis>
    ^goal <goal>)
  (<dis> ^dis-status engage
    ^participant <title>)
  (<goal> ^argument1 good-enough-for
    ^argument2 advisor
    ^argument3 yes
    ^performative answer
    ^sender <title>
    -^status)
-->
  (<s> ^operator <o> + =)
  (<o> ^name answer-good-enough-for-advisor-yes
    ^participant <title>)
}

sp {agent*apply*answer-good-enough-for-advisor-yes
  (state <s> ^operator <o>
    ^agent <agent>)
  (<o> ^name answer-good-enough-for-advisor-yes
    ^participant <participant>)
  (<agent> ^discourse <dis>
    ^emotion <emotion>
    ^goal <ag-goal>)
  (<ag-goal> ^argument1 good-enough-for
    ^argument2 advisor
    ^argument3 yes

```

```

    ^performative answer
    ^sender <title>)
-->
(<agent> ^goal <goal>
  ^goal <goal2>
  ^emotion <emotion> -
  ^emotion good
  ^move-master go)
(<goal> ^argument1 thanks
  ^content |Thank you friend.|
  ^out ok
  ^performative inform
  ^receiver <participant>)
(<goal2> ^content |I decided to end conversation.|
  ^out ok
  ^performative end-discourse
  ^receiver <participant>
  ^status complete)
(<ag-goal> ^status complete)
(<agent> ^discourse <dis> -)
}

#friend
#####
sp {agent*propose*answer-quality-work
  (state <s> ^agent <agent>)
  (<agent> ^discourse <dis>
    ^goal <goal>)
  (<dis> ^dis-status engage
    ^participant <title>)
  (<goal> ^argument1 work
    ^argument2 quality
    ^performative ask
    ^sender <title>
    -^status)
-->
  (<s> ^operator <o> + =)
  (<o> ^name answer-quality-work
    ^participant <title>)
}

sp {agent*apply*answer-quality-work
  (state <s> ^operator <o>
    ^agent <agent>)
  (<o> ^name answer-quality-work
    ^participant <participant>)
}

```

```

(<agent> ^goal <ag-goal>)
(<ag-goal> ^argument1 work
  ^argument2 quality
  ^performative ask
  ^sender <title>)
-->
(<agent> ^goal <goal>)
(<goal> ^argument1 work
  ^argument2 quality
  ^argument3 good
  ^content |The quality of your work is good.|
  ^out ok
  ^performative answer
  ^receiver <participant>)
(<ag-goal> ^status complete)
}

sp {agent*propose*answer-good-enough-for-advisor
  (state <s> ^agent <agent>)
  (<agent> ^discourse <dis>
    ^goal <goal>)
  (<dis> ^dis-status engage
    ^participant <title>)
  (<goal> ^argument1 good-enough-for
    ^argument2 advisor
    ^performative ask
    ^sender <title>
    -^status)
-->
  (<s> ^operator <o> + =)
  (<o> ^name answer-good-enough-for-advisor
    ^participant <title>)
}

sp {agent*apply*answer-good-enough-for-advisor
  (state <s> ^operator <o>
    ^agent <agent>)
  (<o> ^name answer-good-enough-for-advisor
    ^participant <participant>)
  (<agent> ^goal <ag-goal>)
  (<ag-goal> ^argument1 good-enough-for
    ^argument2 advisor
    ^performative ask
    ^sender <title>)
-->
  (<agent> ^goal <goal>)

```

```
(<goal> ^argument1 good-enough-for
  ^argument2 advisor
  ^argument3 yes
  ^content |I believe it is.|
  ^out ok
  ^performative answer
  ^receiver <participant>)
(<ag-goal> ^status complete)
}
```

Selected productions 4. Scripted conversation between graduate student and his or her friend.

APPENDIX C – C++ application that uses the Soar markup language

```
#include <stdlib.h>
#include <string>
// For method intreceiverString
```

```

#include <sstream>
#include <fstream>
#include <iostream>
using namespace std;
#include "sml_Client.h"
using namespace sml;

string print = "print --depth 10 s1";
fstream file;
fstream file1;

/* Convert int receiver string */
string intToString(int integer){
    ostringstream stream;
    stream << integer << flush;
    return stream.str();
}

void agent(Agent* agent, Identifier* pID, Identifier* pCommand){
    string argument1;
    string argument2;
    string argument3;
    string content;
    string envi = pCommand->GetParameterValue("envi");
    string goalStatus;
    string performative = pCommand->GetParameterValue("performative");
    string receiver = pCommand->GetParameterValue("receiver");
    string sender = pCommand->GetParameterValue("sender");

    if(pCommand->GetParameterValue("argument1")){
        argument1 = pCommand->GetParameterValue("argument1");
    }
    if(pCommand->GetParameterValue("argument2")){
        argument2 = pCommand->GetParameterValue("argument2");
    }
    if(pCommand->GetParameterValue("argument3")){
        argument3 = pCommand->GetParameterValue("argument3");
    }
    if(pCommand->GetParameterValue("content")){
        content = pCommand->GetParameterValue("content");
    }
    if(pCommand->GetParameterValue("goal-status")){
        goalStatus = pCommand->GetParameterValue("goal-status");
    }
    // Or could do the same manually like this:
    // mithos->CreateStringWME(pCommand, "status", "complete");

    // Send information
    if(pCommand->GetParameterValue("argument1")){
        agent->CreateStringWME(pID, "argument1", argument1.c_str());
    }
}

```

```

    if(pCommand->GetParameterValue("argument2")){
        agent->CreateStringWME(pID, "argument2", argument2.c_str());
    }
    if(pCommand->GetParameterValue("argument3")){
        agent->CreateStringWME(pID, "argument3", argument3.c_str());
    }
    if(pCommand->GetParameterValue("content")){
        agent->CreateStringWME(pID, "content", content.c_str());
    }
    agent->CreateStringWME(pID, "envi", envi.c_str());
    if(pCommand->GetParameterValue("goal-status")){
        agent->CreateStringWME(pID, "goal-status", goalStatus.c_str());
    }
    agent->CreateStringWME(pID, "performative", performative.c_str());
    agent->CreateStringWME(pID, "receiver", receiver.c_str());
    agent->CreateStringWME(pID, "sender", sender.c_str());
}

void discourse(Agent* agent, Identifier* pID, Identifier* pCommand) {
    // Add attribute that indicates who has the word
    string disStatus = pCommand->GetParameterValue("dis-status");
    string envi = pCommand->GetParameterValue("envi");
    string sender = pCommand->GetParameterValue("sender");
    string receiver = pCommand->GetParameterValue("receiver");
    cout << "(" << sender << " ^dis-status " << disStatus << ")\n";
    cout << "(" << sender << " ^envi " << envi << ")\n";
    cout << "(" << sender << " ^receiver " << receiver << ")\n";
    //string status = pCommand->GetParameterValue("status");
    //cout << "(" << name << " ^status " << status << ")\n\n";
    // Update environment here receiver reflect agent's command
    // Then mark the command as completed
    pCommand->AddStatusComplete();
    // Or could do the same manually like this:
    // mithos->CreateStringWME(pCommand, "status", "complete");

    // Discourse proposal
    agent->CreateStringWME(pID, "dis-status", disStatus.c_str());
    agent->CreateStringWME(pID, "envi", envi.c_str());
    agent->CreateStringWME(pID, "sender", sender.c_str());
    agent->CreateStringWME(pID, "receiver", receiver.c_str());
}

Agent* createAgent(Kernel* pKernel, string name){
    // Create a Soar agent named name
    Agent* agent = pKernel->CreateAgent(name.c_str());
    // Check that nothing went wrong
    // NOTE: No agent gets created if there's a problem, so we have receiver check for
    // errors through the kernel object.
    if (pKernel->HadError())
    {
        cout << pKernel->GetLastErrorDescription() << endl;
    }
}

```

```

        return 0;
    }
    return agent;
}

int loadRules(Agent* agent, string path){
    agent->LoadProductions(path.c_str());
    if (agent->HadError())
    {
        cout << agent->GetLastErrorDescription() << endl;
        return -1;
    }
    return 0;
}

void initAgent(Agent* agent, string name, string content, string performative, string agEnvi,
string emotion, string title, string work, string* envi, int arrayLength){
    Identifier* pInputLink = agent->GetInputLink();
    Identifier* pID = agent->CreateIdWME(pInputLink, "init");
    Identifier* pIDagent = agent->CreateIdWME(pID, "agent");
    agent->CreateStringWME(pIDagent, "name", name.c_str());
    agent->CreateStringWME(pIDagent, "content", content.c_str());
    agent->CreateStringWME(pIDagent, "performative", performative.c_str());
    agent->CreateStringWME(pIDagent, "envi", agEnvi.c_str());
    agent->CreateStringWME(pIDagent, "emotion", emotion.c_str());
    agent->CreateStringWME(pIDagent, "title", title.c_str());
    agent->CreateStringWME(pIDagent, "work", work.c_str());
    Identifier* pIDenvi;
    for(int x=0; x<arrayLength; x++){
        pIDenvi = agent->CreateIdWME(pID, "envi");
        agent->CreateStringWME(pIDenvi, "name", envi[x].c_str());
    }
    agent->RunSelfTilOutput();

    // Destroy information from input-link
    agent->DestroyWME(pID);
}

Identifier* deleteInputLinkInfo(Agent* agent, Identifier* pID, string link){
    Identifier* pInputLink = agent->GetInputLink();
    agent->DestroyWME(pID);
    return agent->CreateIdWME(pInputLink, link.c_str());
}

void communicate(Agent* agent1, Agent* agent2, Identifier* pID){
    // Go through all the commands we've received (if any) since we last ran Soar.
    int numberCommands = agent1->GetNumberCommands();
    for (int i = 0; i < numberCommands; i++)
    {
        Identifier* pMessage = agent2->CreateIdWME(pID, "message");
        Identifier* pCommand = agent1->GetCommand(i);
    }
}

```

```

        string name = pCommand->GetCommandName();
        if(name.compare("agent") == 0)
        {
            agent(agent2, pMessage, pCommand);
        }
    }
}

void printAgent(Identifier* pCommand){
    string envi = pCommand->GetParameterValue("envi");
    string sender = pCommand->GetParameterValue("sender");
    string content = pCommand->GetParameterValue("content");
    string goalStatus;
    string performative = pCommand->GetParameterValue("performative");
    if(pCommand->GetParameterValue("goal-status")){
        goalStatus = pCommand->GetParameterValue("goal-status");
    }
    string receiver = pCommand->GetParameterValue("receiver");
    string argument1;
    if(pCommand->GetParameterValue("argument1")){
        argument1 = pCommand->GetParameterValue("argument1");
    }
    string argument2;
    if(pCommand->GetParameterValue("argument2")){
        argument2 = pCommand->GetParameterValue("argument2");
    }
    string argument3;
    if(pCommand->GetParameterValue("argument3")){
        argument3 = pCommand->GetParameterValue("argument3");
    }
    cout << "(" << sender << " ^envi " << envi << ")\n\t";
    cout << "(" << sender << " ^performative " << performative << ")\n\t";
    cout << "(" << sender << " ^content " << content << ")\n\t";
    file << "(" << sender << " ^envi " << envi << ")\n\t";
    file << "(" << sender << " ^performative " << performative << ")\n\t";
    file << "(" << sender << " ^content " << content << ")\n\t";
    file1 << "(" << sender << " ^envi " << envi << ")\n\t";
    file1 << "(" << sender << " ^performative " << performative << ")\n\t";
    file1 << "(" << sender << " ^content " << content << ")\n\t";
    if(pCommand->GetParameterValue("goal-status")){
        cout << "(" << sender << " ^goal-status " << goalStatus << ")\n\t";
        file << "(" << sender << " ^goal-status " << goalStatus << ")\n\t";
        file1 << "(" << sender << " ^goal-status " << goalStatus << ")\n\t";
    }
    if(pCommand->GetParameterValue("argument1")){
        cout << "(" << sender << " ^argument1 " << argument1 << ")\n\t";
        file << "(" << sender << " ^argument1 " << argument1 << ")\n\t";
        file1 << "(" << sender << " ^argument1 " << argument1 << ")\n\t";
    }
    if(pCommand->GetParameterValue("argument2")){

```

```

        cout << "(" << sender << " ^argument2 " << argument2 << ")\\n\\t";
        file << "(" << sender << " ^argument2 " << argument2 << ")\\n\\t";
        file1 << "(" << sender << " ^argument2 " << argument2 << ")\\n\\t";
    }
    if(pCommand->GetParameterValue("argument3")){
        cout << "(" << sender << " ^argument3 " << argument3 << ")\\n\\t";
        file << "(" << sender << " ^argument3 " << argument3 << ")\\n\\t";
        file1 << "(" << sender << " ^argument3 " << argument3 << ")\\n\\t";
    }
    cout << "(" << sender << " ^receiver " << receiver << ")\\n";
    file << "(" << sender << " ^receiver " << receiver << ")\\n";
    file1 << "(" << sender << " ^receiver " << receiver << ")\\n";
    // Update environment here receiver reflect agent's command
    // Then mark the command as completed
    pCommand->AddStatusComplete();
}

void communicatePrint(Agent* agent){
    // Go through all the commands we've received (if any) since we last ran Soar.
    int numberCommands = agent->GetNumberCommands()-1;
    for (int i = 0; numberCommands >= 0; numberCommands--, i++)
    {
        Identifier* pCommand = agent->GetCommand(numberCommands);
        string name = pCommand->GetCommandName();
        cout << intToString(i) << ".\\t";
        file << intToString(i) << ".\\t";
        file1 << intToString(i) << ".\\t";
        if(name.compare("agent") == 0)
        {
            printAgent(pCommand);
        }
    }
}

int main(){
    string state;
    file.open ("Lukis Tales with state.txt", ios::out | ios::trunc);
    if (!file.is_open())
    {
        return -1;
    }
    file1.open ("Lukis Tales without state.txt", ios::out | ios::trunc);
    if (!file1.is_open())
    {
        return -1;
    }
    //file << "<html><head><title>Lukis Tales</title></head><body>";

    // Create an instance of the Soar kernel in our process
    Kernel* pKernel = Kernel::CreateKernelInNewThread();
    // Check that nothing went wrong. We will always get back a kernel object

```

```

// even if something went wrong and we have receiver abort.
if (pKernel->HadError())
{
    cout << pKernel->GetLastErrorDescription() << endl;
    return -1;
}

Agent* ag1 = createAgent(pKernel, "ag1");
Agent* ag2 = createAgent(pKernel, "ag2");
Agent* ag3 = createAgent(pKernel, "ag3");
Agent* ag4 = createAgent(pKernel, "ag4");

if(!ag1||!ag2||!ag3||!ag4)
    return -1;

// Load some productions
string path = string(pKernel->GetLibraryLocation()) + "/Tests/Lukis-Tales.soar";
if(loadRules(ag1,path)==-1)
    return -1;
path = string(pKernel->GetLibraryLocation()) + "/Tests/Lukis-Tales.soar";
if(loadRules(ag2,path)==-1)
    return -1;
path = string(pKernel->GetLibraryLocation()) + "/Tests/Lukis-Tales.soar";
if(loadRules(ag3,path)==-1)
    return -1;
path = string(pKernel->GetLibraryLocation()) + "/Tests/Lukis-Tales.soar";
if(loadRules(ag4,path)==-1)
    return -1;

int arrayLength = 5;
string* envi;
envi = new string[arrayLength];
envi[0] = "apartment";
envi[1] = "lab";
envi[2] = "street";
envi[3] = "coffee-house";
envi[4] = "class";

Identifier* pIDag1 = ag1->CreateIdWME(ag1->GetInputLink(), "agent");
Identifier* pIDag2 = ag2->CreateIdWME(ag2->GetInputLink(), "agent");
Identifier* pIDag3 = ag3->CreateIdWME(ag3->GetInputLink(), "agent");
Identifier* pIDag4 = ag4->CreateIdWME(ag4->GetInputLink(), "agent");

bool init = true;
int x =0;
// Communication
while(true && x!=150){
    x++;
    if(init){
        // Initialize characters

```

```

        initAgent(ag1, "ag1", "I need to graduate.", "graduate", "apartment",
"good", "graduate-student", "bad", envi, arrayLength);
        initAgent(ag2, "ag2", "I have nothing to do.", "none", "street", "good",
"friend", "none", envi, arrayLength);
        initAgent(ag3, "ag3", "I have nothing to do.", "none", "street", "good",
"enemy", "none", envi, arrayLength);
        initAgent(ag4, "ag4", "I must not lose my graduate student.", "keep-
graduate-student", "apartment", "good", "advisor", "none", envi, arrayLength);
        init = false;
    }

```

```
// Print S1
```

```
//file << "<font color=""blue"">";
```

```
cout << "=====\nag1\n";
```

```
file << "=====\nag1\n";
```

```
state = ag1->ExecuteCommandLine(print.c_str());
```

```
cout << state;
```

```
cout << "\nag2\n";
```

```
file << state;
```

```
file << "\nag2\n";
```

```
state = ag2->ExecuteCommandLine(print.c_str());
```

```
cout << state;
```

```
cout << "\nag3\n";
```

```
file << state;
```

```
file << "\nag3\n";
```

```
state = ag3->ExecuteCommandLine(print.c_str());
```

```
cout << state;
```

```
cout << "\nag4\n";
```

```
file << state;
```

```
file << "\nag4\n";
```

```
state = ag4->ExecuteCommandLine(print.c_str());
```

```
cout << state;
```

```
file << state;
```

```
//file << "</font><font color=""red"">";
```

```
cout << "\n===Agent talk=== ag1 says:\n";
```

```
file << "\n===Agent talk=== ag1 says:\n";
```

```
file1 << "\n===Agent talk=== ag1 says:\n";
```

```
communicate(ag1, ag2, pIDag2);
```

```
communicate(ag1, ag3, pIDag3);
```

```
communicate(ag1, ag4, pIDag4);
```

```
communicatePrint(ag1);
```

```
cout << "===Agent talk=== ag2 says:\n";
```

```
file << "===Agent talk=== ag2 says:\n";
```

```
file1 << "===Agent talk=== ag2 says:\n";
```

```
communicate(ag2, ag1, pIDag1);
```

```
communicate(ag2, ag3, pIDag3);
```

```
communicate(ag2, ag4, pIDag4);
```

```
communicatePrint(ag2);
```

```

cout << "===Agent talk=== ag3 says:\n";
file << "===Agent talk=== ag3 says:\n";
file1 << "===Agent talk=== ag3 says:\n";
communicate(ag3, ag1, pIDag1);
communicate(ag3, ag2, pIDag2);
communicate(ag3, ag4, pIDag4);
communicatePrint(ag3);
cout << "===Agent talk=== ag4 says:\n";
file << "===Agent talk=== ag4 says:\n";
file1 << "===Agent talk=== ag4 says:\n";
communicate(ag4, ag1, pIDag1);
communicate(ag4, ag2, pIDag2);
communicate(ag4, ag3, pIDag3);
communicatePrint(ag4);
//file << "</font>";

cout << "=====\nInput:\n";
file << "=====\n";
file1 << "=====\n";

// Run all agents
ag1->RunSelfTilOutput();
ag2->RunSelfTilOutput();
ag3->RunSelfTilOutput();
ag4->RunSelfTilOutput();

// delete info on Input-Link
pIDag1 = deleteInputLinkInfo(ag1, pIDag1, "agent");
pIDag2 = deleteInputLinkInfo(ag2, pIDag2, "agent");
pIDag3 = deleteInputLinkInfo(ag3, pIDag3, "agent");
pIDag4 = deleteInputLinkInfo(ag4, pIDag4, "agent");

/*string quit;
cin >> quit;
if(quit.compare("p") == 0)
    break;*/
}

// Create an example Soar command line
std::string cmd = "excise --all";

// Execute the command
char const* pResult = pKernel->ExecuteCommandLine(cmd.c_str(), "ag1");
pResult = pKernel->ExecuteCommandLine(cmd.c_str(), "ag2");
pResult = pKernel->ExecuteCommandLine(cmd.c_str(), "ag3");
pResult = pKernel->ExecuteCommandLine(cmd.c_str(), "ag4");

// Shutdown and clean up
pKernel->Shutdown(); // Deletes all agents (unless using a remote connection)
delete pKernel; // Deletes the kernel itself

```

```
//file << "</body></html>";  
file.close();  
file1.close();  
  
return 0;  
}
```