Language Technologies in Speech-Enabled Second Language Learning Games: From Reading to Dialogue

By

Yushi Xu

S.M. in Electrical Engineering and Computer Science (2008)

Massachusetts Institute of Technology

Submitted to Department of Electrical Engineering and Computer Science

in Partial Fulfillment

of the Requirements for the Degree of Ph.D

at the

Massachusetts Institute of Technology

June, 2012

Signature of the author [illegible]

Department of Electrical Engineering and Computer Science

February 28, 2012

Certified by [illegible]

Stephanie Seneff

Senior Research Scientist

Thesis Supervisor

Accepted by [illegible]

Leslie A. Kolodziejski

Chair, Department Committee on Graduate Students

Language Technologies in Speech-Enabled Second Language Learning Games:

From Reading to Dialogue

By

Yushi Xu

Submitted to Department of Electrical Engineering and Computer Science
In February 28, 2012 in Partial Fulfillment
of the Requirements for the Degree of Ph.D
in Electrical Engineering and Computer Science

ABSTRACT

Second language learning has become an important societal need over the past decades. Given that the number of language teachers is far below demand, computer-aided language learning software is becoming a promising supplement to traditional classroom learning, as well as potentially enabling new opportunities for self-learning. The use of speech technologies is especially attractive to offer students unlimited chances for speaking exercises. To create helpful and intelligent speaking exercises on a computer, it is necessary for the computer to not only recognize the acoustics, but also to understand the meaning and give appropriate responses. Nevertheless, most existing speech-enabled language learning software focuses only on speech recognition and pronunciation training. Very few have emphasized exercising the student's composition and comprehension abilities and adopting language technologies to enable free-form conversation emulating a real human tutor.

This thesis investigates the critical functionalities of a computer-aided language learning system, and presents a generic framework as well as various language- and domain-independent modules to enable building complex speech-based language learning systems. Four games have been designed and implemented using the framework and the modules to demonstrate their usability and flexibility, where dynamic content creation, automatic assessment, and automatic assistance are emphasized. The four games, reading, translation, question-answering and dialogue, offer different activities with gradually increasing difficulty, and involve a wide range of language processing techniques, such as language understanding, language generation, question generation, context resolution, dialogue management and user simulation. User studies with real subjects show that the systems were well received and judged to be helpful.

Thesis supervisor: Stephanie Seneff

Title: Senior Research Scientist

Acknowledgments

I would like to express my sincere gratitude to my supervisor Stephanie Seneff, who has given me guidance and enormous encouragements. Her earnest in research and kindness as a supervisor and a colleague strongly influence me during my Master's and Ph.D courses, and will continue having impact on me in a longer term. I would also like to thank my committee, Regina Barzilay and Victor Zue, for giving me helpful suggestions on the thesis.

This thesis would not be completed without my collaborators and colleagues. I would like to thank Scott Cyphers and Lee Hetherington for offering all kinds of hardware and software support. I would like to thank Ian McGraw for his generous help on WAMI. I would like to thank Anna Goldie for her ideas and effort in helping creating the question-answering game. I would like to thank both Ian and Anna for appearing in the video demo of the game systems. My gratitude also goes to all those who participated in the user study, and/or gave comments and suggestions about the systems.

I would like to thank Jim Glass and all those who are or were a member of Spoken Language Systems Group during my course of study. I would like to thank Marcia Davidson for providing various administrative assistance. I would like to thank my lovely officemates and ex-officemates, Ibrahim Badr, Carrie Cai, Han-pang Chiu, Ann Lee, Jingjing Liu, and Rabih Zbib for creating such a nice and enjoyable office environment. I would also like to thank Hung-an Chang, Chia-ying Lee, Ekapol Chuangsuwanich, Yaodong Zhang, and other students in the SLS group for academic discussions and social activities.

I am thankful to all my friends, both in China and in the US, who have offered me help and shared my joy and upset.

Finally, I would like to give my gratitude to my parents, who have been supporting their only daughter to come to the other side of the world to pursue graduate study. Without their love and vision, I would not have established my first interest in computer science in my childhood, and later entered this renowned institute.

This thesis was supported by MIT, and partially by Quanta Computer, Inc. through the T-Party Initiative.

Table of Contents

List of Figures

List of Tables

URLs of Demonstrative Videos

Reading Game: http://people.csail.mit.edu/seneff/scill/reading_game.wmv

Translation Game: http://people.csail.mit.edu/seneff/scill/translation_game.wmv

Question-Answering Game: http://people.csail.mit.edu/seneff/scill/Q&A.wmv

Dialogue Game: http://people.csail.mit.edu/seneff/scill/dialogue_game.mp4

Chapter 1 Motivation and Background

1.1 Motivation

Computers have dramatically changed many aspects in our lives. Since the popularization of personal computers during the 1990's, these electronic machines have been shrinking in size, gaining more capabilities in computation and memory, and, becoming more intelligent. The ubiquitous existence, not only as desktops and laptops, but also as smart phones, game consoles, *etc.*, together with the cheaply available Internet, has led to a new style of casual life. If people want to shop, they go to online boutiques; if they want to meet with friends, they dial a video call from their computer; if they want to cook a new dish, they find recipes provided by people all over the world; but if they want to learn a language, they probably go into a physical classroom.

This is not to say that computers have not been helping out in the area of education, more specifically, in the area of language learning, but the changes are very limited. Back in the early 90's when software was delivered in floppy disks, there already existed software that helped people prepare tests for various subjects. These programs usually loaded multiple-choice exercises from a database, and allowed the user to type A, B, C or D, but fancier versions for middle school geometry, for example, even allowed the action of adding auxiliary lines in the figures. In those days, it was very exciting to use these kinds of software, because personal computers were still a rare thing at that time, and because they provided a way to eliminate the necessity of turning the pages of the exercise books back and forth to check the answers.

Viewed from today, it is obvious that there was little technology in those kinds of software. Nevertheless, looking at the language learning software in today's market, Rosetta Stone, one of the bestsellers, or EnglishTown.com, the largest online language learning institute, or other smaller online or offline software, most are simply a prettier version of those text-based ones run in the DOS era. They certainly have colorful interfaces, cute animations, and carefully designed contents, but the exercises are still in the styles of multiple choice and fill in the blanks, of which the answers can be judged by a simple string match.

When speech technology came to the commercial market, many realized that it would be a new attractive feature to include speech recognition in the software. It is, however, an attractive gimmick, rather than a useful modality, and hardly intelligent, for students are not expected to speak other than the pre-designed answer, and nothing beyond a string match is performed. No wonder people still prefer to attend face-to-face classes, if their goals are not only remembering vocabulary and grammar, but to be able to effectively use the language.

But the need is so great. Take Mandarin Chinese for example. According to the Chinese Ministry of Education, more than 30 million people worldwide were learning Mandarin Chinese in 2007, and the number would reach a hundred million by 2010 [**1**]. Another report has predicted the number to approach 150 million by 2013 [**2**]. The same report estimated a global shortage of over 5 million Chinese language teachers, while in comparison, only 10 thousand teachers in China possess the teaching license on Chinese as a foreign language. This is an opportunity for language learning computer systems, as well as a big challenge, for simply providing electronic texts and multiple choices is not satisfactory. It can, perhaps, replace the traditional textbooks, but not any human tutors, and people believe computers should be able to do much more than that.

And they should. The advancement in artificial intelligence has been proving many possibilities, including understanding human languages. The algorithms may not be clever enough yet to be compared with humans, but they surely provide a way to go much farther than to do string comparisons. What is missing is a flexible framework that can accommodate various speech and language processing modules and clearly define the functionality of each of them, as well as those modules, which are designed within a language-learning context, while at the same time generic enough to be shared among different systems.

This is where this thesis is situated. We will discuss the roles that the computer, or the artificial intelligence algorithms, should take, leading to a three-layer conceptualization that describes intelligent interactive systems for language learning. The framework will be utilized to show a series of four Chinese learning games, *reading*, *translation*, *quesiton-answering*, and *dialogue*, where each sets the foundation for the next one. Critical modules that solve various kinds of technical difficulties are developed. We extend the template-based content generation approach to allow bilingual templates and lesson-like organizations. In the translation game, we improve the existing paradigm for generating reference translation and meaning comparison to handle inputs from more generic domains. In the question-answering game, new approaches are proposed to generate questions from statements, and to judge the student's answer by using context resolution.

In the dialogue game, a new entity-constraint-based dialogue manager is designed for goal-directed mixed-initiative dialogues. A user simulator that incorporates personality model is also proposed and implemented. We also design algorithms to generate appropriate dialogue scenarios and to provide comprehensive assessment.

We conduct user studies for the four games with real subjects. We will describe the experiments and include discussions on the results found in the studies and experiments.

1.2 Background

The world of computer-aided language learning (CALL) seems to be separated into two sides: the commercial side and the research side. The two sides are like two separate continents, each developing software and technologies in vastly different aspects with minimal communication with each other.

1.2.1 Commercial CALL systems

On the commercial side, the focus of the companies are whole systems that can be deployed on a large scale. These systems used to be delivered via CD-ROMs, and nowadays they are moving towards Internet-based online systems, such as Rosetta Stone Online [**3**], English Town [**4**], Fluenz [**5**], and Chengo Chinese [**6**]. These systems usually have nicely designed lesson materials and user interfaces. The typical scenario of a lesson is that, the student is first shown a text in the format of paragraphs, animations or videos, followed by a list of important vocabulary and grammar points. After that, manually prepared exercises are presented. These exercises are mostly static and objective: multiple choices, pair matching, filling in the blanks *etc.*, and if not, the students' answer is usually not graded. Popping up a message saying "you have completed this exercise" is all it does. Most of the recent software has incorporated speech technologies. However, when speaking exercises are selected, the system either expects an exact speech input from the student, or sets up a video conference connection between the student and a human teacher.

Large companies usually have strong teams responsible for creating the texts and exercises, investigating students' behaviors and progress, and creating new pedagogically effective ideas, which make their software rich in context and strong in design. New speech and language technologies appear in their advertisements, but they are more like a future direction than the reality. Through interactions with a large language learning software company, we noticed that the speech recognition technologies were used in very primitive ways, and engineers in the company did not have a good understanding of the speech recognition engine. Despite their willingness to take on new technologies, they were merely ignorant of the existence of the whole research continent opposite their own continent.

1.2.2 Research works

Computer-aided language learning is a relatively new interdisciplinary field. The mix of pedagogy, linguistics and computer science creates many different sub-areas for researchers. As many see speaking ability as the bottleneck for traditional classroom teaching, it rises naturally to use automatic spoken language technology, which has been researched for over four decades, to improve students' speaking performance. Pronunciation training thus becomes a prevalent research topic. A considerable number of papers have been published which experimented with different models to predict students' pronunciation proficiency, and to give feedback. Some examples are Goodness of Pronunciation which calculates the likelihood ratio between the actual pronounced phone and the canonical pronunciation [7], extended recognition network with phonologically motivated rules that model common pronunciation errors [8][9], and the novel structure-based model which assesses the structure formed by all the vowels as a whole instead of every single vowel [10]. These effort, focus on phone level errors. Their goal is to teach a nonnative speaker to pronounce the sounds as closely to a native speaker as possible. Other research has investigated sentence level fluency and intonation [11][12]. A system emphasizing prosody is exemplified by [13] which integrates pitch, phones, timing and stress together. But in all of these works, the measurements are purely acoustic. The syntax and vocabulary learning are implicitly contained in the materials that the students read aloud, and the systems do not pay attention to the verbal content of the students' speech.

It is arguable that poor pronunciation is the major factor that hinders the students' speaking ability. Even for native speakers, people have accents due to their living environment. Spontaneous speech is a combination of language composition and speech production in a short time, and thus we consider the composition ability to have equal importance with the pronunciation. Nevertheless, in the research area, compared to the amount of research done in pronunciation training, much less has been done that involves composition ability. Part of the reason might be, compared to the solid achievement in speech representation, the modeling of semantics still remains a difficult research problem. There exist, however, a few systems that try to target composition ability. The flight-reservation domain translation game previously developed in our group is an example. In a translation activity, the students have to express the meaning using their own words, and spoken translation specifically involves both sentence composition and speech production. In this translation game, students are expected to give spoken translation in Chinese of the displayed English sentences related to flight reservations. The game is very domain specific. It uses a domain-specific grammar to understand the student's speech, and evaluates the translation based on the domain-specific meaning slots such as source city and destination city.

HELEN **[14][15]** is another system that exercises students' composition ability from a more comprehensive aspect. In this system, authors can input texts, and the system automatically generates comprehension tests from the text materials. Semantics is involved, so that appropriate interrogative words can be picked to form questions. Rather than multiple choices, the system expects answers in natural sentences. The system supports speech synthesis to play back the questions, but speech recognition is not available. Students must enter the answers through text input.

Researchers from Kyoto University also developed a system that requires students to form their own sentences **[16]**. The system asks the student questions, and the student needs to answer according to the pictures he sees on the screen. The questions and the sentence patterns that students practice are based on the lesson points, and the speech recognizer includes models that try to predict possible errors.

Spoken translation and spoken question-answering provide better exercises than simple reading, but neither of them is the ultimate exercise people desire for language learning. Students need to speak more: not to read aloud some given text, but to speak spontaneously; and not to speak isolated utterances, but to talk in a context. This leads to the obvious answer: spoken dialogue systems.

The research field of general spoken dialogue systems emerged about two decades ago. People develop systems in the hope that they can replace human operators in some service industries. In recent years, statistical methods have gained popularity in the field. Markov decision processes and Partially Observable Markov decision processes (POMDP) have been the focus of a number of papers **[17][18][19][20]**. These approaches turn the dialogue interaction strategy into an optimization problem. The dialogue manager selects the actions prescribed by the optimal policy; *i.e.*, the one that maximizes the reward function **[21]**. This machine learning formulation of the problem enables the automation of system development. The models are learnt from large amounts of conversational data. Developers are thus freed from hand-coded rules, which require expertise and are usually difficult to design to cover the entire space. In the Spoken Dialogue Challenge (SDC) 2010 [**22**] in which the participants implemented the "LET'S GO" dialogue systems to provide bus information in Pittsburg, U.S., a dialogue system using POMDP is demonstrated [**23**]. Another statistical dialogue system [**24**] demonstrated in the same challenge adopted a belief state tracking strategy over multiple dialogue states to achieve more robustness to speech recognition errors.

However, looking into the spoken dialogue systems for language learning, the situation is different. The prevalence of data-driven methods and machine learning has reached various aspects of spoken language processing, but dialogue management for language learning is one exception. Three reasons can be considered to account for the situation. First, obtaining data is difficult. Machine learning algorithms rely highly on the amount and quality of the data. The most preferable data for dialogue management used in language learning would be the conversations between a nonnative speaker and a native speaker. Apparently, it is very hard and expensive to obtain such kinds of data, and it might be even harder and more expensive to clean and annotate all sorts of errors nonnative speakers make.

The second reason is the system behavior. Data-driven statistical dialogue systems relieve developers from rule-writing, but on the other hand, they keep the human farther from understanding and controlling their behavior. In a statistical framework, to alter the output might involve modifying the reward function, the objective function, state transition matrix, *etc.* [**25**]. The opaqueness of how the change of one parameter will affect the final output is very unattractive to language teachers. Better control is preferred in the educational setting, because language teachers usually have specific pedagogical objectives embedded in the dialogues, which should be presented in a rather faithful way.

Third, users of general spoken dialogue systems and students using the CALL systems have very distinct purposes. The former group uses the systems for obtaining information or getting assistance. They are usually tolerant of minor system errors. For the CALL systems, however, errors are a serious issue. Students have high expectations of the systems' accuracy, since they themselves do not have enough knowledge to distinguish the incorrect from the correct in the new language. Under the assumption that the systems always teaches the correct usage of the language, errors in the systems would mislead the students, which betrays the purposes of teaching.

Therefore, the dialogue management for language learning systems takes very different approaches which seem out-of-date in the trend of "statisticalization". The simplest case is the scripted dialogue model, which is now largely used in most of the commercial systems. The system displays a prompt at each turn, and asks the student to speak the prompt. A later model shows multiple prompts, but still only one of them is correct. The dialogue proceeds when the student has successfully spoken the correct candidate. Another variety is that no candidate is displayed. The student has to compose a response by himself. But since there still exists a unique correct answer, oftentimes the activity becomes a guessing game and frustrates the students.

An improved version in the research area used a finite state dialogue network instead of the one-route dialogue, as adopted in *Subarashii*, a system that teaches Japanese [**26**][**27**]. Instead of one unique correct response, multiple responses are acceptable at a given state, and each transits the dialogue to a different state. This makes the dialogue more interesting and less scripted, but as a trade-off, the amount of human effort in creating the dialogue structure increases. In the *Subarashii* system, an interface was created to collect pivot dialogue data in text mode. Authors then use the data to create appropriate finite state networks that model those dialogues. Even though the four dialogues in the system were very simple, the initial evaluation showed that about half of the students' input fell out of the designed networks.

Scripted dialogue systems are easily manageable by non-experts. They also have an advantage in high speech recognition accuracy, because the answer space is extremely limited. Similarly, no real-time language processing is required, for the system only matches the recognition hypotheses against the response candidates.

However, the flexibility of the allowed inputs is greatly constrained in the scripted dialogue systems. Researchers have been exploring ways to allow freer inputs so that the system's behavior is more human-like, which requires reliable speech recognition, language understanding and dialogue management modules. To assure high accuracy and good controllability, the dialogue management is usually rule-based; and to further lower the complexity, the dialogues are usually goal-directed, and are commonly confined to a pre-defined domain for a given system. Researchers at CMU have developed a telephone-based bus schedule dialogue system *LET'S GO* [**28**], and adapted it to language learning [**29**]. The system uses a generic dialogue management framework *RavenClaw* [**30**], which separates the architecture into a domain-dependent "Dialog Task Specification" layer and a domain-independent "Dialog Engine". A tree of dialogue agents, each of which handles different dialogue actions, is specified to control the dialogue progress. For language learning purposes, the dialogue system is capable of giving feedback on incorrect inputs, assuming that a list of correct inputs exists beforehand. The incorrect input is aligned with the closest correct input, and the difference is emphasized through speech synthesis.

Another language learning dialogue system is the *EqualParty* system [31], originally developed under the *GALAXY* framework [32], which used ordered system actions along with a set of trigger conditions as a control script. Systems developed using *GALAXY* include the flight domain *MERCURY* [33] and weather domain *JUPITER* [34], and *EqualParty* was specially designed for language learning purposes. The system is called "EqualParty" because there is a symmetry between the human and the computer sides of the communication, in contrast to systems where the computer plays the role of an agent. The system generates a persona and time schedule for the student, and asks the student to negotiate with the system to find a common time for an activity that they both "like". The system maintains a very specific dialogue component that handles dialogue turns in this domain. One interesting design of the system is that it divides the learning procedure into five stages. Assuming the student has no previous knowledge about the language, the first stage is to eavesdrop on a dialogue between two computer parties, followed by the second stage of parroting one computer party. As the student gradually progresses, in the final stage, he will be able to conduct a dialogue with the computer completely on his own. This five-stage design not only serves as a strategy for language learning, but also helps students learn about the system and know what is expected, which constrains the inputs from the user side.

Wik and his colleagues have been developing animated avatars to aid Swedish learning. In their system *Ville* [35], an avatar teacher guides, encourages and gives feedback on pronunciation and perception exercises. As a free-standing part of Ville, they also have developed a dialogue system called *DEAL* [36]. The system sets up a flea-market scenario with visual displays of the merchandise. The student's task is to negotiate with the shopkeeper to buy a set of specified items using the money given. The input is processed through *Higgins* [37], a spoken dialogue system that includes modules for semantic interpretation and analysis. The dialogue action is chosen according to a set of simple rules based on episodic knowledge structure, and certain heuristic haggling strategies. The uniqueness of this system is that it calls on the student's observation skills, as pointing out the defects of the merchandise would lead to a lower price.

There are many other sub-areas in computer-assisted language learning. Eskenazi has written an informative review paper on spoken language technology for education, and specifically for language learning [38]. Most of the work reveals that spoken language technologies for language learning is a separate research area from general spoken language research. Simply applying the methods developed for general spoken language application does not suffice, when it comes to nonnative and erroneous inputs and the demand of highly accurate outputs. It also reveals that very few systematic frameworks exist for semantically-involved whole systems. Some systems are adapted from systems built for native speakers, and most only provide one type of activity, which should not be true, if the modules in the system are truly generic and they are organized in a non ad-hoc way.

1.3 Thesis outline

The rest of the thesis is organized as follows:

Chapter 2 Computer as a Learning Partner: In this chapter, the functionalities that a computer system should have in a language learning setting are elucidated. In what aspects can the computer help us? How can we put the functionalities together in a reasonable and flexible way? The discussion leads to a framework at the end of the chapter.

Chapter 3 Template-Based Content Creation: Preparing content is a huge job. We try to move from traditional flat text-based contents to a more compact representation. The representation also allows the content creator to encode two languages with different word orders in the same file.

Chapter 4 The First Three Games: Reading, Translation and Question-Answering: This chapter discusses implementations of full systems, as well as individual components. Using the framework described in Chapter 2, we design and implement games in which the difficulty level rises gradually, both for the students and for the developers. Three games are included in the chapter, alongside their underlying generic modules: parsing, language generation, meaning rewrite, and context resolution. The details of these modules will be explained, together with how they are utilized to fulfill the language learning functionalities. The chapter concludes with system evaluation and a user study.

Chapter 5 Entity-Constraint-Based Dialogue Management: Dialogue management is an indispensible module for a dialogue game. This chapter introduces a newly designed entity-based dialogue manager. Applications in the flight-reservation domain and the drug domain are included.

Chapter 6 The Fourth Game: Dialogue: A template-based user simulator and the module for performance assessment are first introduced in this chapter. A dialogue game is built using these modules. Based on the real data we collected, methods for improving system performance are discussed.

Chapter 7 Summary and Future Directions: A final discussion is given.

Chapter 2 Computer as a Learning Partner

2.1 Functionalities of a language learning system

Before we begin, we should first ask one basic question. What do we expect the computer to do in the language learning setting? The answer is rather obvious, and is agreed on by both the commercial leaders and the researchers: to replace the human teachers. But this answer is very idealistic. While we are talking about language learning systems, nobody will imagine, at least in the near future, a school with a computer room and no language teachers to offer English lessons, or a self-study student to master the speaking, listening, reading and writing abilities solely by learning with a computer. People clearly realize that the current technologies are not yet adequate to simulate human behavior in all aspects. That is why the field is call computer-*assisted* language learning, rather than computer-*based* language learning. Thus, a more realistic answer to our question would be, to replace the human teachers *to some extent.*

But to what extent? What is the best we can do? To answer this question, we need to first examine the teacher's job. The human language teacher's job can be summarized into three aspects: preparing core materials, presenting content, and answering questions.

By core materials, we mean the key grammar points and vocabulary items, sometimes including a set of example sentences, but not including the exercises. These materials need to be divided into lessons with increasing difficulty at a reasonable pace, and thus the preparation requires great understanding of both linguistics and pedagogy. The authors of the textbooks usually take the larger part in the preparation, and the rest is managed by the teacher in the classroom. This aspect is almost fully occupied by human intelligence, and there is very little attempt to have computers involved besides as a typing machine. The core materials presented to the students need to be very precise and well-selected. It is not surprising that people regard computers as incompetent in this aspect, since this is far beyond the current advancement of artificial intelligence.

Once the core materials are designed, the best way to present them depends on the teacher who actually interacts with the students, and thus varies a lot from teacher to teacher. However, exercises are the common method that most teachers adopt. Well-designed exercises can implicitly cover new sentence patterns and vocabulary, and can help students memorize the new materials.

Computer systems have great potential in this aspect. Not only can they display the exercises created by a human, they should also be capable of generating exercises by themselves, once the underlying vocabulary and sentence patterns are given. The students can complete the exercises by typing. But since speaking is the ability that many language teachers emphasize and cannot be practiced alone, the most rewarding exercise would be having the student answer by speech. Given that spoken language technologies have been improving over the years, the computer should be able to handle students' various speech input in natural sentences.

Automatic grading is the other task that comes with exercise generation, for exercise without grading does not make a lot sense. If the computer can create exercises, judge the correctness, and give verbal feedback, it then makes the interaction more similar to a classroom situation where the teacher calls on a student to finish an oral exercise. There should not be only one single form of exercise, either, for different activities have different degrees of emphasis, which can help improve students' language proficiency in different ways.

Providing and assessing exercises is not the end of the story. As the ancient Chinese scholar Han stated "a teacher is the person who transmits wisdom, imparts knowledge and resolves doubts,"[**39**] being able to answer students' questions is another crucial responsibility of a teacher, and failure to do so by computers is one major reason why people perceive human teachers as a better teaching resource than computers. This completely dynamic activity is truly a difficult task for computer systems. Extremely high-level language understanding must be involved. Nevertheless, if we simplify the problem a bit, it is possible to have the computers provide clues or hints dynamically in the exercises. Creating proper scaffolding will make the computer systems more person-like, and make the students less frustrated.

To summarize, in order to make a computer system more similar to a human teacher, and to lessen the manual effort by the largest degree, the computer should have the following three functionalities in a single activity: dynamic exercise creation, *i.e.*, generating dynamic exercise problems using the given core materials; performance assessment, *i.e.*, judging the correctness of the answers in one exercise and monitoring the student's performance over time; and assistance, *i.e.*, giving appropriate clues and hints when the student gets stuck. There is a fourth functionality implied by these three functionalities: language understanding, whether spoken or written, so that the activities will have natural language interfaces. The student can then see the exercise in natural language, complete it using spoken natural language, and receive feedback in (spoken) natural language.

The fifth functionality is that multiple types of activities are desired, such as reading exercises, translation exercises, and dialogue exercises. The framework should be flexible to extend existing activities and add new activities, so that different language abilities are practiced, and the students do not get bored by a single type of exercise.

There are other aspects that help the computer become more human-like, such as incorporating visual input and building human-like robot teachers. These are beyond the scope of this thesis, and will not be discussed.

2.2 Speech-enabled architecture with WAMI

Having discussed the desired functionalities, this section considers the architecture of a system that can potentially fulfill the functionalities. Such a system involves several components, the speech recognizer, the speech synthesizer, the graphical user interface (GUI) and the language processing component. To integrate them together, we use the *WAMI* toolkit [**40**]. This toolkit provides easy application programming interfaces (API) to create speech-enabled Web-based application. It has been successfully used to build several speech-enabled applications, such as *CityBrowser* [**41**], *WordWar* [**42**], *Rainbow Rummy* [**43**], as well as to enhance existing web applications with speech capability, such as the *Voice Race* [**44**] and *Voice Scatter* [**45**] games provided by *Quizlet* [**46**].

Under the WAMI architecture, the speech recognizer, speech synthesizer, and language processing component run on the server side. The user interface on the client side is typically a webpage containing a WAMI applet. As illustrated in Figure 2-1, WAMI mediates the communication of the components. A typical turn starts with the applet on the user interface capturing the input speech and transmitting it to WAMI. WAMI sends the waveforms to the recognizer, receives an N-best recognition hypothesis list, and sends it to the language processing component. The language processing component then produces a response based on the input. If the response contains a speech synthesis instruction (synthesis dispatch), WAMI communicates with the speech synthesizer, and sends the synthesized waveforms to the client for playback. If the response contains a user interface update (GUI dispatch), WAMI updates the webpage on the client side with proper information. Text inputs and mouse clicks on the user interface are captured and processed in similar ways, except that the speech recognizer is not involved in this case.

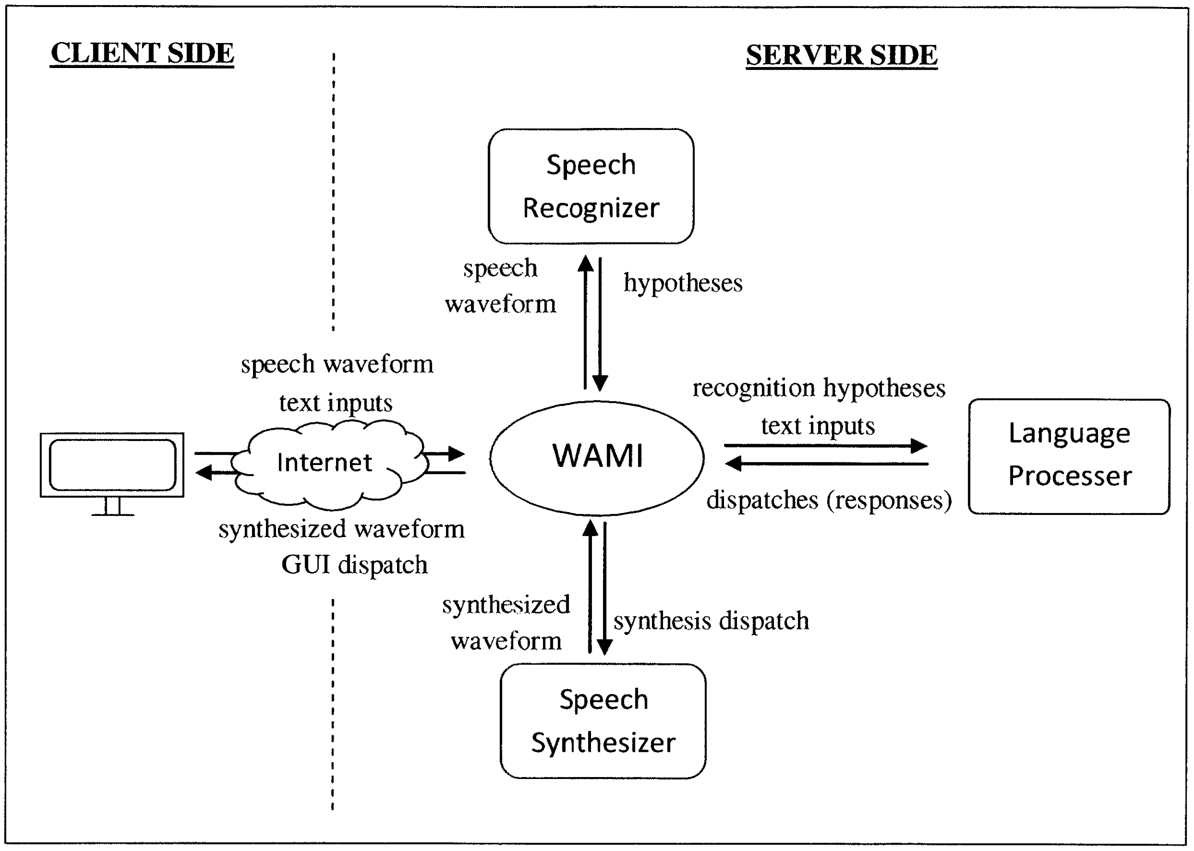


Figure 2-1. The system architecture with WAMI, including the client side, the speech recognizer, the speech synthesizer and the language processer on the server side.

2.3 Three-layer conceptualization

Under the WAMI architecture, the speech recognizer and the speech synthesizer have very specific functionalities. The functionalities of the language processor, however, vary from activity to activity. For example, in a reading activity, the language processing is as simple as string comparison, while in a dialogue activity, complex language processing is necessary to understand the meaning of the input and generate an appropriate response. To support multiple activities and have them share similar processing steps, the entire language processor needs to be broken down into smaller modules. The modules are most desired to be generic in all three dimensions: language, domain and activity. Language independence enables the possibility of handling both the student's first language (L1) and the language to be learnt (L2), as well as porting the activity to other language pairs. Domain independence allows the activity to be extended to other domains of interest without much effort, for example, from a travel domain translation activity to a restaurant domain translation activity. Activity independence enables the re-use of the modules in different activities. For example, a language understanding module can be shared by a translation activity and a dialogue activity.

Nevertheless, since the final system implements a language learning activity that has specific behaviors for a specific second language, the independence criteria cannot be satisfied by all the modules. To maximally conform to the independence criteria, we propose a conceptualization that groups the modules into three layers: the language layer, the meaning layer, and the language tutor layer, as shown in Figure 2-2.

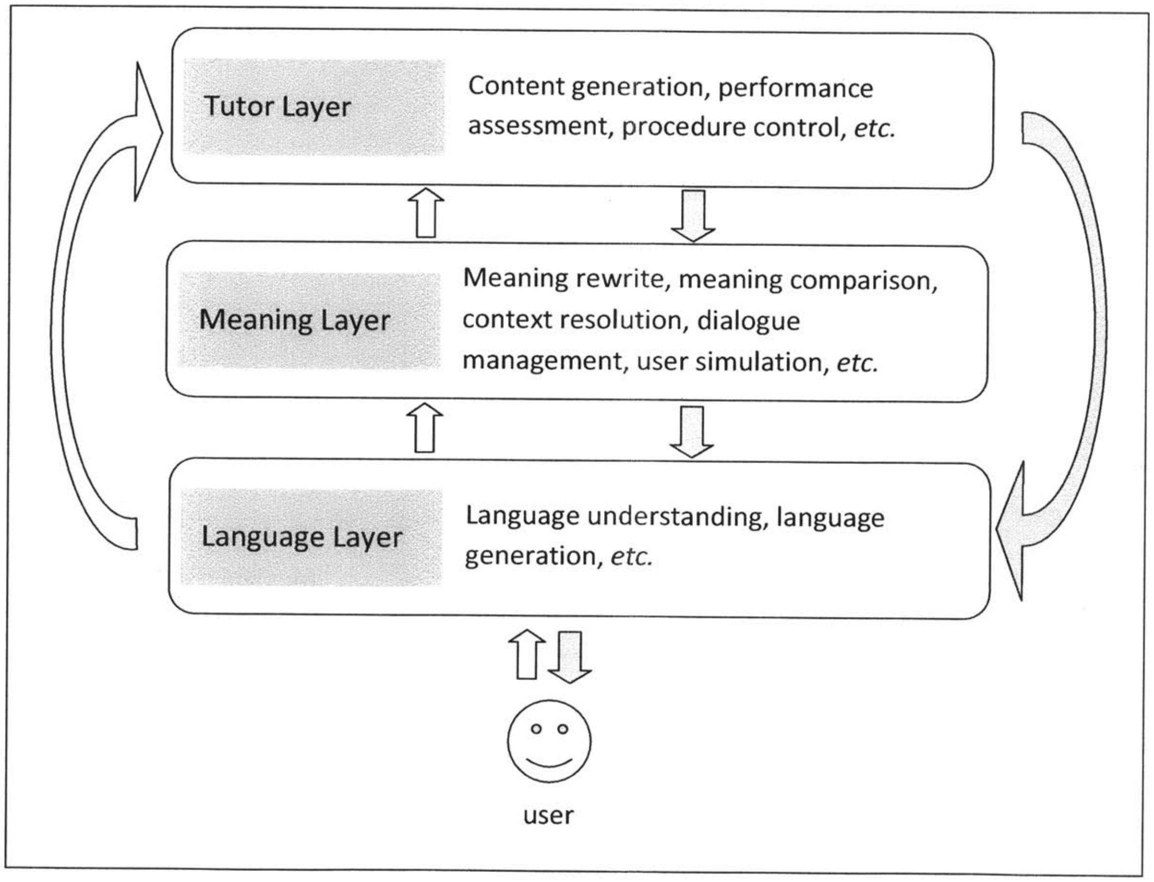

Tutor Layer
Content generation, performance assessment, procedure control, *etc.*
Meaning Layer
Meaning rewrite, meaning comparison, context resolution, dialogue management, user simulation, *etc.*
Language Layer
Language understanding, language generation, *etc.*
user

Figure 2-2. Three-layer conceptualization consisting of the language layer, the meaning layer and the tutor layer from bottom to top. The language layer serves as a natural language interface. The meaning layer provides semantic processing. The tutor layer handles application-specific behavior and provides language learning features.

The language layer sits on the bottom. The modules on this layer interact with the natural sentences in a specific language, and converts them from/to meaning representations. Typical modules are language understanding (parsing) and language generation. Though the modules deal with sentences in a particular language, the modules themselves can be both language- and domain independent, with the rule files specific in the language, and/or in the domain. This layer is the closest to the user. It creates the natural language interface for the user, and can be used in many activities.

In the middle is the meaning layer. The modules on this layer, such as meaning rewrite, meaning comparison, context resolution, dialogue management and user simulation, work with meaning representations, and thus are language-independent. The domain dependency again can be enforced by domain-specific configuration files, so that the modules remain domain independent. As the modules on this layer usually provide general language processing functionalities and are not designed for one particular language learning activity, they can be potentially used by different activities in a flexible way.

Table 2-1. Independence properties of the language layer, the meaning layer and the tutor layer.

Layer		Language Independent	Domain Independent	Activity Independent
Tutor Layer	Modules	Maybe	Maybe	No
	External rules/configuration	Maybe	Maybe	No
Meaning Layer	Modules	Yes	Yes	Yes
	External rules/configuration	Yes	No	Yes
Language Layer	Modules	Yes	Yes	Yes
	External rules/configuration	No	No	Yes

As we can see, the language layer and the meaning layer are general-purpose; they can be used to create systems for purposes other than language learning. The top layer, the tutor layer, covers modules that are uniquely required for language learning systems. The modules include content creation, performance assessment, as well as procedure control of the activity. These modules are mostly activity-dependent, and some are domain- and language-dependent. Table 2-1 summarizes the independence properties of the three layers.

2.4 Implementation with turn manager

The three-layer conceptualization is implemented using a turn management framework. This framework evolved from the GALAXY II framework [47], in which a central hub communicates with various components in the system, including a turn management component, which in turn runs from a set of dialogue control rules. The new turn manager consolidates the hub and the former turn management, and uses a control scripting language to replace both the hub's PGM (program) script and the original turn manager's control rules.

In the turn management framework, a turn is defined as the period between when the turn manager receives a message and when it sends out the return message. Usually, the message is a representation of the user's input, and the return message is the system's response. The message is processed through a list of operations provided by various modules, with an order dictated by the dialogue control (DCTL) script. The turn manager maintains two shared states: a turn state that transmits information between operations in one turn, and a session state which stores global variables of an entire session.

```
{ c statement
      :topic { q pronoun
             :name "this" }
      :pred { p copular
             :topic { q frame
                    :quant "indef" } } }
```

Figure 2-3. An example Galaxy frame, which represents the semantic meaning of the sentence "this is a frame."

Both the control script and the turn/session state are coded in the Galaxy frame format. Figure 2-3 shows an example Galaxy frame. Details of the format can be found in [47]. In short, a

Galaxy frame is enclosed by a set of curly braces, and starts with its type followed by its name. The type can be one of the following three types: "c" for clause, "p" for predicate, and "q" for topic (quantified set). The content of the frame includes two types of elements: predicates and keys. A predicate is a p-type frame led by a "*:pred*". Multiple predicates are allowed in one Galaxy frame. A key has a name and a value. The name can be any string that starts with a colon except the special key, "*:pred*", and the value can be an integer, a string, a double, another Galaxy frame, or a list of any of the above value types. In the example, "*:topic*" is a key of the statement frame, and "*:name*" is a key of the pronoun frame. Only one instance of a given key can appear in a frame. In the rest of the thesis, the term "frame" will be used to denote Galaxy frames.

```
{c dctl
   :initial_frame {c frame
         :domains "newPhrasebook"
         :languages "english" }
   :rules ( {c rule
              :variables {c variables
                    :rules ":parse_and_paraphrase" }
              :operation "subroutine_call" } )
   :parse_and_paraphrase (
            {c rule
              :conditions ":input_string"
              :variables {c variables
                    :input_key ":input_string"
                    :domain "newPhrasebook"
                    :output_key ":parse_frame" }
              :operation "create_frame" }
            {c rule
              :conditions ":parse_frame"
              :variables {c variables
                    :input_key ":parse_frame"
                    :domain "newPhrasebook"
                    :language "english"
                    :output_key ":paraphrase_string" }
              :operation "paraphrase_frame" } ) }
```

Figure 2-4. An example DCTL script to parse and paraphrase an input sentence.

```
:rules (
   {c rule
      :conditions ":x"
      :variables {c variables
             :rules ":then" }
     :operation "subroutine_call" }
   {c rule
      :conditions "!:x"
      :variables {c variables
             :rules ":else" }
     :operation "subroutine_call" } )
:then (
      ...
      ... )
:else (
      ...
      ...)
```

(a) Branch

```
:rules (
   {c rule
      :variables {c variables
             :set ":counter 0" }
      :operation "nop" }
   {c rule
      :variables {c variables
             :rules ":loop" }
      :operation "subroutine_call" } )
:loop (
   ...
   ...
   {c rule
      :variables {c variables
             :counter_key ":counter" }
      :operation "increment_counter" }
   {c rule
      :conditions ":counter < 10"
      :variables {c variables
             :rules ":loop" }
      :operation "subroutine_call" } )
```

(b) Iteration

Figure 2-5. Branch and iteration using the DCTL script.

The turn state and session state are represented by two frames. Each key in the frame denotes a variable. A new frame for the session state is created upon a new session. It maintains all the global variables, as well as a finite history list of some important state variables. A new frame for the turn state is created in the beginning of each turn, and the entire frame is destroyed at the end of the turn.

The control script DCTL consists of two parts: an "initial frame" and lists of ordered rules. Figure 2-4 gives an example DCTL script which parses and paraphrases an input sentence. The initial frame specifies the arguments for the turn manager. A full list of the valid arguments can be found in Appendix A.

The rules are grouped into several lists, with the root list led by the key *:rules*. When a turn starts, rules in the root list are executed sequentially. Each rule contains three basic elements: *:conditions*, *:variables*, and *:operation*, which indicate the conditions to trigger the operation, the arguments of the operation and the name of the operation. An operation can be viewed as a handler provided by a specific module. Some special control operations initiate subroutine calls, which load another list of rules led by the specified key, and run the rules in the new list sequentially. Upon finishing, the program will restore the original rule list and the position that has been run through, and start from the next rule.

The DCTL script provides a powerful and flexible capability to organize the operations to perform desired functionalities. Like traditional programming languages, branch and iteration can be realized, as illustrated in Figure 2-5. The framework also allows rule lists in one DCTL file to be shared with a second file using a key *:insert_files* in the top level of the second file. This capability allows multiple systems to share common subsets of rules defining macros.

2.5 Summary

In a language learning system, the computer needs to fulfill three functionalities: creating exercises, assessing student's performance, and giving proper assistance. To automate all these three aspects, deep speech and language processing is indispensable. Furthermore, the processing needs to be properly modularized, so that new activities can take advantage of the existing modules to reduce the development time.

We use the WAMI toolkit to construct the overall system architecture with a Web-based user interface, speech recognizers, speech synthesizers and a language processor. Within the language processor, a three-layer conceptualization is introduced to abstract the processing procedure. The language layer takes in a natural text input, which can be the recognition hypothesis, and converts it into a meaning representation. The meaning layer then performs processing on the meaning level, such as meaning comparison, context resolution and dialogue management. The result of the meaning processing enters the tutor layer for application-specific handling, such as game control and performance assessment. The system's responses then go down to the meaning layer and the language layer to be transformed into natural sentences. Components on different layers have different language-, domain- and activity independent properties. In short, components on the language and meaning layers are easily sharable among different language learning activities. The tutor layer contains components that tend to be activity-dependent.

The three-layer conceptualization is implemented in the turn manager framework. In this framework, handlers are provided by various modules as operations, and a scripting language is utilized to instruct the turn manager of the order of the operations. The scripts consist of conditioned rules organized into lists, and this makes the framework much easier to maintain than its ancestor framework. The operations and modules are easily extendable as well, so that different systems can have their unique behavior by combining existing common modules and application-specific modules.

Having discussed the architecture and the three-layer conceptualization, now we will begin to discuss the modules on the three layers in more detail. We will start with content creation in the next chapter.

Chapter 3 Template-Based Content Creation

The contents of the exercises are the meat of any language learning system. Although new technologies in speech and language can help more students learn better and faster, they are merely equations and algorithms without contents, and will not transfer into any real systems.

Contents come from various sources. Existing textbooks can be a good source. Alternatively, many commercial companies and research groups ask experienced language teachers to create contents specially designed for the particular systems. In either way, a reasonable representation is necessary to store the contents in computer files for system access. This chapter discusses using templates to represent the contents. Several improvements are proposed for the template-based approach to handle non-context-free content generation and bilingual content generation. Methods to generate contents in a lesson mode which gradually introduces new materials are also discussed.

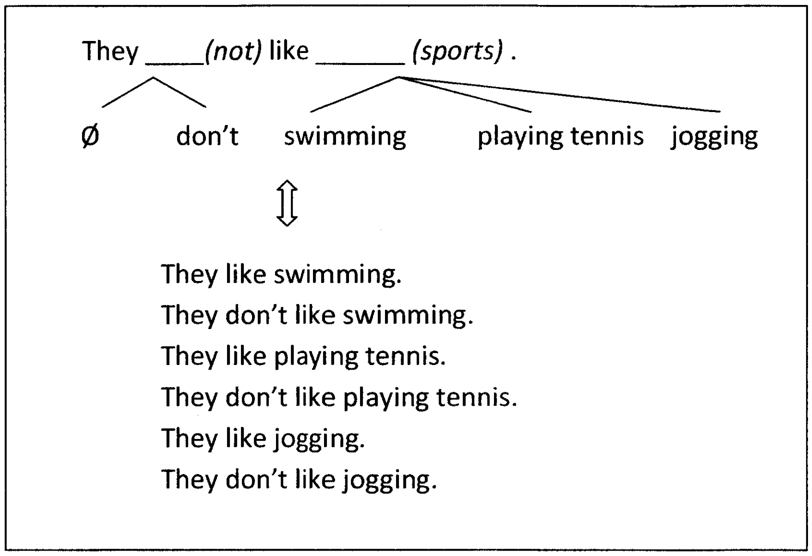
3.1 Template representation of contents

In many language learning activities that focus on sentence structure and composition ability, contents appear in the form of phrases and sentences. For example, students are asked to read sentences in the reading activity, or to translate sentences in the translation activity. The sentences contain important grammar points, which are usually sentence patterns, as well as vocabulary of a specific level. One can imagine the formation of such sentences as filling the vocabulary into the sentence patterns, and thus templates have been used to represent the contents in some of the previous systems [**48**] [**49**].

Figure 3-1 shows an illustration of the template representation. The template on the top represents a sentence pattern. The two slots in the template are associated with different vocabulary choices. This template is able to generate six distinct sentences.

The template representation is not restricted to a two-level hierarchy. Sub-templates can be introduced to create intermediate levels. The sub-templates and the vocabulary can then be shared by multiple super-templates, which makes the templates similar to a context-free grammar (CFG).

To encode such multi-level templates, we use the GALAXY frame format, shown in Figure 3-2. In this format, each key defines a sub-template list, and a special key *:templates* introduces the root template list. Within each template, tokens starting with a colon indicate a slot, which should be instantiated with a corresponding sub-template.



They ____*(not)* like ______ *(sports)*.
Ø
don't
swimming
playing tennis
jogging
They like swimming.
They don't like swimming.
They like playing tennis.
They don't like playing tennis.
They like jogging.
They don't like jogging.

Figure 3-1. A template representation of content consisting of two levels.

```
{c template
     :templates (
          "They :not like :sports ."
          ":ball_sports is his favorite sport."
     )
     :not ( "" "don't" )
     :sports ( "swimming" "playing :ball_sports" "jogging" )
     :ball_sports ("tennis" "soccer" "basketball" )  }
```

Figure 3-2. Multi-level templates written using the GALAXY frame format.

This template representation is compact and simple for non-experts to compose. The root template list can be instantiated into an exponential number of unique sentences via straightforward algorithms, which can be used to generate both the exercise contents and training corpora for language models. However, this simple representation has its limitations. In the next two sections, we discuss improvements over the simple template representation to achieve a more powerful representation.

3.2 Non-CFG templates

As mentioned above, the templates resemble a context-free grammar, *i.e.*, the instantiations of the slots in a template are independent of each other. Nevertheless, the context-free property prohibits the representation of certain contents. Consider the following two templates.

I was born in :city. In other words, :city is my hometown.
I'm going to :city from :city .

In the first template, the two occurrences of *:city* should be instantiated into the same city, while in the second template, they should be different cities. To produce the desired sentences, a context is necessary during the instantiation to record the instantiation of each sub-template. To distinguish the equivalent instantiation, *i.e.* the first case, and the distinct instantiation, *i.e.* the second case, a syntax is created such that equivalent instantiation is adopted by default, and the distinct instantiation is denoted by a one-digit suffix of the slot. According to the syntax, the distinct instantiation is expressed as follows. Note that *:city1* and *:city2* are both instantiated from the same sub-template list, with the added constraint that they cannot be the same value.

I'm going to :city1 from :city2 .

We also observe the difficulty in capturing the correct inflections and agreements in languages like English, due to the hierarchical structure. To provide an easy fix for these templates, a set of rewrite rules are allowed to perform final amendments on the instantiated sentences. Figure 3-3 exemplifies two solutions to produce sentences with correct subject-verb agreement by creating separate sub-templates for singular verbs, and by using the rewrite rules.

Following is the algorithm outline to generate a sentence from the non-context-free templates. A context is created and used to record all existing slot instantiations for the template in order to handle the equivalent instantiations and distinct instantiations.

```
1. Create an instantiation context c.
2. Randomly pick a template t from the root template list.
3. Find the first slot s in t, and instantiate it according to context c; or if s does not exist
   in c, replace s with a randomly item v from the corresponding sub-template list, remove
   v from the sub-template list, and record (s,v) in c.
4. Repeat Step 3 until no slot is left in t.
5. Apply rewrite rules on t.
```

```
{c template
      :templates (
            " :pronoun is a student."
            " :pronoun_third :verb_sg ."
            " :pronoun_other :verb ."    )
      :pronoun ( " :pronoun_other " " :pronoun_third " )
      :pronoun_other ( "I" "You" )
      :pronoun_third ( "He" "She" )
      :verb ( "walk" "run" "sing" "dance" )
      :verb_sg ( "walks" "runs" "sings" "dances" )
      :rewrite_rules (  "I is"    "I am"
                        "You is"   "You are" ) }
```

Figure 3-3. An example of utilizing separate sub-templates and rewrite rules to fix verb inflections.

3.3 Bilingual templates

Oftentimes when preparing the contents, bilingual contents are useful. Consider a reading exercise where the student is asked to read a list of sentences in L2. It is useful if the L2 sentences are paired with L1 sentences, so that the students can easily be provided with a translation of what they are reading. Bilingual sentence pairs can also be used directly to create activities such as pair matching, where the sentences in two languages are shown on two sides of the screen, and the students need to match those with equivalent meaning together. Therefore, it would be helpful if the templates can encode contents in two languages, and instantiate bilingual sentence pairs simultaneously.

The straightforward idea is to replace the monolingual strings in the template with bilingual strings. We use a vertical bar as a separator between the two languages, and parentheses for appropriate grouping. Figure 3-4 is a bilingual version of the templates in Figure 3-2. These templates can generate the following sentence using the algorithm described in the last section.

(They | 他们) (don't | 不) (like | 喜欢) (playing | 打) (soccer | 足球) .

A bilingual sentence pair can be easily formed by picking apart the left and right item of each group. After applying the rewrite rules to the left language and the pair rewrite rules to the right language, we have the following sentence pair.

```
{c template
      :templates (
          " ( They | 他们 ) :not ( like | 喜欢 ) :sports ."
          ":ball_sports ( is | 是 ) ( his favorite sport | 他最喜欢的运动 ) ."
      )
      :not ( "" "( don't | 不 )" )
      :sports ( " ( swimming | 游泳) "  " (playing | 打 ) :ball_sports"
                " ( jogging | 跑步 ) " )
      :ball_sports (" ( tennis | 网球 ) "  "( soccer | 足球 ) "
                    " ( basketball | 篮球) " )
      :pair_rewrite_rules ( " 打 足球"  "踢 足球" ) }
```

Figure 3-4. A bilingual version of the templates in Figure 3-2.

```
{c template
      :templates (
          " ( They | 他们 ) :time_R ( won the game | 赢 了 比赛 ) :time_L ."
      )
      :time ( "yesterday | 昨天"  " last week | 上周") }
```

Figure 3-5. An example of bilingual templates with word order differences.

They don't like playing soccer.
他们 不 喜欢 踢 足球.

This syntax assumes the two languages follow the same word order, which is rarely true. For example, the underlined temporal phrases in the following pair appear at different positions.

They won the game yesterday .
他们 昨天 赢 了 比赛.

We need a way to express the different word orders that is both easy and effective. Suffixes *_L* and *_R* are introduced to solve the problem.

The two suffixes stand for the instantiation of only the left/right language for this blank, leaving the other side empty. As shown in Figure 3-5, *:time_L* and *:time_R* are put in different positions. The immediate instantiation result would be the following sentence.

(They | 他们) (| 昨天) (won | 赢 了) (the game | 比赛) (yesterday |) .

After picking apart the two sides, we have the sentence pair with correct word orders.

The syntax with *_L* and *_R* is very simple, yet adequate to describe complex reordering between the two languages. Figure 3-6 illustrates an example which can generate sentence pairs with very different word orders.

```
{c template
      :templates (
            " :adjunct_L ( He | 他 ) :adjunct_R ( wants | 想 ) :to_do"
      )
      :adjunct ( " :conj_L :action :conj_R (, | )" )
      :conj ( " after | 后 " " before | 前 " )
      :action  ( " graduation | 毕业 " " work | 工作 " )
      :to_do ( " (to | ) :to_place_R  ( travel  | 旅游) :to_place_L " )
      :to_place ( " (to | 去) ( China | 中国) " ) }
```

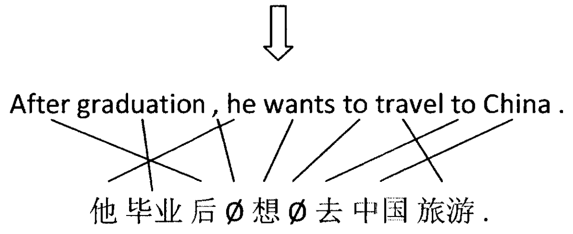

After graduation , he wants to travel to China .
他 毕业 后 ∅ 想 ∅ 去 中国 旅游 .

Figure 3-6. A more complex example of bilingual templates with word order differences.

3.4 Templates with lessons

3.4.1 Organizing templates into lessons

Having the ability to generate both monolingual sentences and bilingual pairs from the templates, now we consider how to improve the organization of the templates and sub-templates. In traditional textbooks or exercise books, materials are usually divided into lessons. Earlier lessons introduce easier contents, and later lessons add more complexity and difficulty. We would like to organize the templates in a similar way, so that, when instantiating the templates, it is possible to control the difficulty of generated sentences, and to have them contain patterns and vocabulary at the appropriate level in a frame.

```
{c template
   :lesson1 {c template
      :templates (
           "They :not like :sports ."
           ":ball_sports is his favorite sport."
       )
       :not ( "" "don't" )
       :sports ( "swimming" "playing :ball_sports" "jogging" )
       :ball_sport ("tennis" "soccer" "basketball" )   }

   :lesson2 {c template
       :templates (
           "He prefers :sports1 to :sports2 ."
       )
       :sports ( "yoga"  "ballroom dancing" )
       :ball_sports ( "baseball"  "volleyball"  "table tennis" ) } }
```

Figure 3-7. A template frame with two lessons.

To implement this idea, we group the templates into lessons, with each lesson having its own root template list. Figure 3-7 shows a template frame with two lessons. The sentence pattern of lesson two is more complex, but the vocabulary overlaps with the previous lesson. This is a common situation. Thus, we designed the algorithm so that later lessons can have access to the sub-template lists introduced in the previous lessons. In addition, later lessons can augment an existing sub-template list. In the example, two more sports and three more ball sports are introduced in the *:sports* and *:ball_sports* lists respectively in lesson two. They will be combined with those already defined in lesson one. Therefore, the following sentence is a possible instantiation of the template in lesson two.

He prefers swimming to playing baseball.

But at the same time, earlier lessons retain their smaller size of vocabulary, and thus it is not possible to generate either of the following sentences from lesson one.

They like yoga.
Table tennis is his favorite sport.

3.4.2 Blending lessons

In real activities, for example translation, a set of exercises consists of multiple sentences. We would want the sentences to be generated from the specified lesson, but sometimes also cover a certain amount of review materials. This section discusses a blending method to achieve this purpose and maximize the coverage of the materials given a fixed number of generated sentences.

Given a lesson K, to generate a sentence, a template is selected from all the possible templates based on the weights defined below

$$w_k = \begin{cases} \alpha^{K-k} & 1 \le k \le K \\ 0 & k > K \end{cases} \tag{3-1}$$

Where k is the lesson index in which the template is defined, and α is a fading coefficient which takes value between 0 and 1. Thus, templates defined in lesson 1 up to lesson K have non-zero weights, and the more recent lessons have larger weights. By adjusting the coefficient α, we are able to control the proportion of review materials we want in the generated sentence set. The weights also apply to the sub-templates, which means that the sub-pattern and vocabulary introduced in more recent lessons are more likely to be chosen as the instantiation than the older ones. This is desired, since we want the student to focus more on the new vocabulary.

On the other hand, given a fixed number of sentences in one exercise, it is preferred that the sentences cover as many patterns and as much vocabulary as possible. We want to avoid the situation where one particular template repeatedly occurs, and the student loses the chance to be exposed to other templates. The algorithm used in previous systems assigns uniform probability distribution over all the templates in one template list. We modified the algorithm to adjust the probabilities according to the number of times the particular template has been chosen.

Equation 3-2 shows the detailed calculation of the probability assigned to template t in a template list L, where β is a coefficient ranging from 0 to 1, and c(•) is the count of template t_i being previously chosen. The more times it has been chosen, the less likely it will be to choose it again, so the counts for all templates will be more balanced.

$$P(t) = \frac{\beta^{c(t)}}{\sum_{t_i \in L} \beta^{c(t_i)}} \tag{3-2}$$

Combining Equation 3-1 and Equation 3-2, we have the complete probability calculation for each template t in template list L, which originally defined in lesson k, when the current lesson is K.

$$P(t) = \frac{w(k)\beta^{c(t)}}{\sum_{t_i \in L} w(k_i)\beta^{c(t_i)}}$$
$$w(k) = \begin{cases} \alpha^{K-k} & 1 \le k \le K \\ 0 & k > K \end{cases} \tag{3-3}$$

When instantiating a template, the sub-templates are drawn according to their probability distribution, so that students are more likely to see patterns and vocabulary they have not seen in previous sentences. In actual applications, coefficients α and β are both set to 0.5. Thus the generation output consists of approximately half review materials and half new materials.

The following gives a brief algorithm for generating *n* sentences from lesson *k* of a template frame.

```
1. Blend the templates from lessons 1 to k.
2. Initialize a count array for every template list.
3. Create an instantiation context c, randomly pick a template from the root template list, and instantiate it with lesson weights and count arrays.
4. Update the count arrays according to the records in context c.
5. Repeat Step 3 and Step 4 for n times.
```

3.5 Summary

Creating contents for language learning systems is no easy job. Small amounts of contents would result in frequent repetition while students use the system, but on the other hand, the task of content creation becomes very tedious when the amount of contents increases. We consider part of this problem to lie in the representation of the contents. With a better representation, the human effort should be able to be reduced.

We use the template representation to encode the contents. Several improvements have been introduced to allow the templates to represent more flexible contents. Non-context free templates are enabled by using an instantiation context and rewrite rules. We also invented a simple left-right suffix syntax to encode bilingual contents in a single template and to efficiently address the word order differences in the two languages.

For use in language learning systems, the templates are organized into lessons. Sub-templates defined in previous lessons are sharable with the later lessons. The instantiation result of a particular lesson consists of materials from this lesson, as well as review materials from previous lessons. The expectation of the ratio of new and review materials is controlled by a fading coefficient. The instantiation algorithm also adjusts the probability distribution of the items in a template list according to the number of times they have been chosen, in order to have the generated sentences cover a larger set of patterns and vocabulary more evenly.

In the next chapter, the templates will be used to provide contents for the language learning systems. Three systems will be introduced: the reading game, the translation game, and the question-answering game.

Chapter 4 The First Three Games: Reading, Translation and Question-Answering

We have talked about the architecture and conceptualization in Chapter 2, and the contents in Chapter 3. This chapter concerns the task of building real systems. But what kind of systems do we want to build? To build systems that help the student learn a second language means two things. First, the system provides educational materials. Second, the system attracts the student. The second point is not trivial, for only if students are motivated to use the system for a long enough period of time can the carefully designed educational materials be of real use. The new technologies behind the system can potentially be of value, but the form in which the students learn is equally important. Presenting the exercises as a test is not very interesting, since tests usually imply stressfulness and strict rules. To make the learning process more fun and relaxing, we consider games as a good way to present the exercises.

In a game, the player's performance is usually measured using game points. Although a score is also produced for tests, the ideas behind the scores for a test and for a game are rather different. The score of a test usually have an upper-bound. If a test is taken twice, the two scores are not additive. However, in many games, there is no upper-bound of the game points. Scores from different game plays can be cumulative, which gives the players motivation to repeat the game – the game points never decrease by playing more. We consider this as important, for motivation is the most important factor [**50**], so that the students would use the system and learn by using.

Another common feature of games is levels. Players are usually confronted with easy tasks in initial levels, and the complexity gradually increases in later levels. This goes well with the design of difficulty-graded lessons.

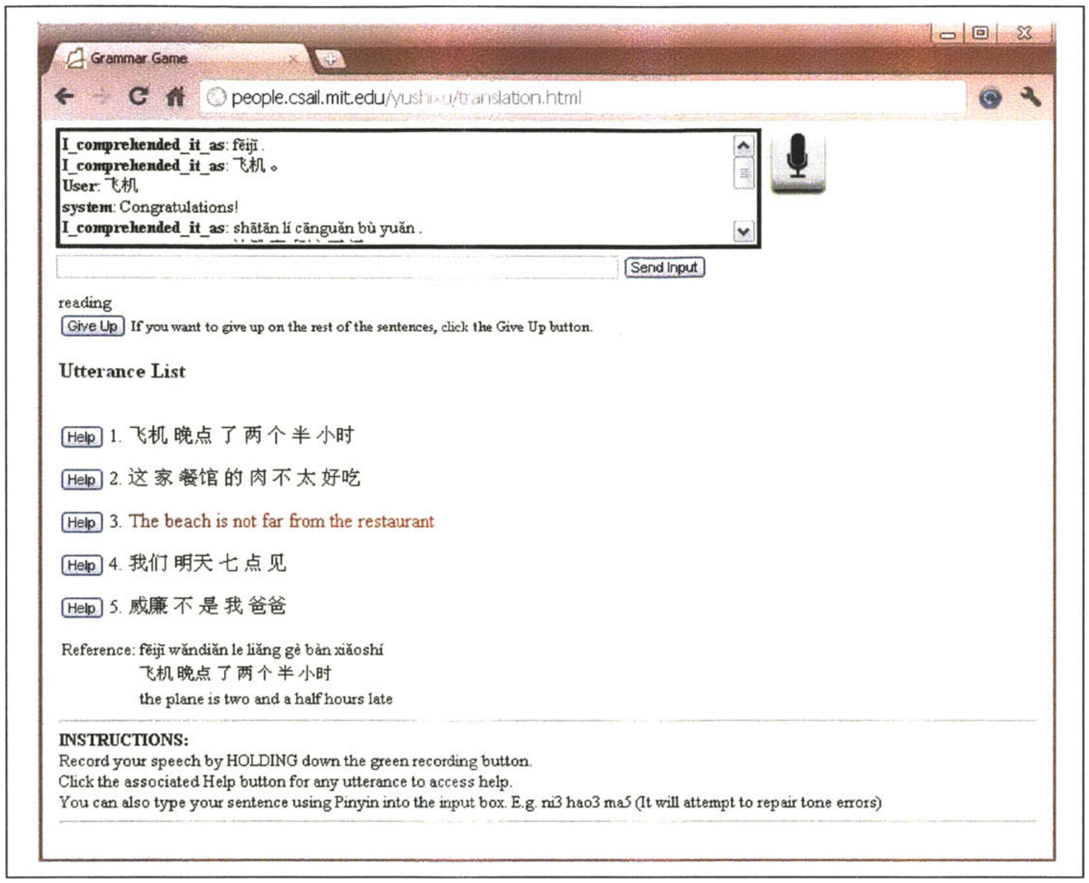
With the game points and levels, performance assessment can be incorporated smoothly. We divide the assessment into two parts. Utterance-level assessment offers judgment, for example, of whether the translation spoken is correct or not, while the overall assessment measures the performance over a longer period of time, *e.g.*, a round or multiple rounds. The game points and levels fit well into the overall assessment: when the student performs well enough for several rounds, he should be able to collect enough points to progress to the next level.

Using this idea, we design a series of language learning games using the architecture described in Chapter 2. All the games are web-accessible and speech-enabled. All the processing is done on the server side. The student only needs a web browser to access the systems.

The games cover a wide variety of activities, and provide exercises for different students from advanced beginner level, *i.e.*, knowing some basic knowledge of the language, to intermediate level, who have two or three years of experience with the language. As mentioned in previous chapters, the games focus on the student's composition and comprehension abilities, rather than pronunciation accuracy. The vocabulary build-up is inherent in the contents; as the student is exposed to more materials, the vocabulary size should increase accordingly.

The four games are designed in such a way that, within each game, multiple difficulty levels are provided, and across the games, a gradual increase in difficulty is provided, where each game prepares the student for the next one. For advanced beginners, we would like the students to learn the pronunciation of the words, and obtain some impression about the sentence structures and vocabulary in the language. Thus, we design a *reading* game as the first game, where the student is only asked to read out aloud the displayed L2 text. When he is able to read sentences fluently, he then moves on to the second game, *translation*, where he needs to come up with the sentences in L2 by himself, by using the prompt in L1. In the third game, *question-answering*, the L1 prompt is further eliminated. Listening exercises are also included. The student is presented with texts in L2, and is required to understand both the texts and the spoken questions in L2, and then provide a spontaneous answer. After the student is able to listen and answer well in the third game, he is challenged in the last game, *dialogue*, to complete a multi-turn dialogue with the system. A dialogue task is given, and natural dialogue interaction is expected with the system to accomplish the task. Successful completion of this game demonstrates that the student is able to communicate on this topic through speech.

In this chapter, we describe the first three games in detail, together with the critical technical solutions to generating reference translations, judging translations, generating questions and judging answers. The dialogue game will be described in the next two chapters. All the games are demonstrated in a setting of Chinese learning for English speakers, but, as we will see, the approaches can be used in other language pairs because of the language independence properties of the modules.



I_comprehended_it_as: fēijī .
I_comprehended_it_as: 飞机 。
User: 飞机
system: Congratulations!
I_comprehended_it_as: shātān lí cānguǎn bù yuǎn .

reading
If you want to give up on the rest of the sentences, click the Give Up button.

Utterance List

1. 飞机 晚点 了 两 个 半 小时
2. 这 家 餐馆 的 肉 不 太 好吃
3. The beach is not far from the restaurant
4. 我们 明天 七 点 见
5. 威廉 不 是 我 爸爸

Reference: fēijī wǎndiǎn le liǎng gè bàn xiǎoshí
飞机 晚点 了 两 个 半 小时
the plane is two and a half hours late

INSTRUCTIONS:
Record your speech by HOLDING down the green recording button.
Click the associated Help button for any utterance to access help.
You can also type your sentence using Pinyin into the input box. E.g. ni3 hao3 ma5 (It will attempt to repair tone errors)

Figure 4-1. A screenshot of the reading game in the middle of a round. The sentence that has been read correctly is marked with red and displayed in English. A demonstrative video is available at http://people.csail.mit.edu/seneff/scill/reading_game.wmv.

4.1 The reading game

4.1.1 Game overview

Our concept of the reading game builds on the assumption that the beginner student has certain initial knowledge of the L2 language. Given a sequence of spellings (Pinyin representations in the case of Mandarin Chinese), the student is able to produce some pronunciations, though not necessarily correct pronunciations. The reading game helps the student practice the pronunciation through listening to the synthesized speech and reading aloud the given text. It also helps the student get a sense of the sentence structures and accumulate simple vocabulary in L2 by providing parallel sentences and grammar points in L1.

Figure 4-1 shows a screenshot of the reading game. The game task is very simple. In each round, the system presents the student a list of sentences which are randomly generated from the corresponding lesson templates, and the student tries to read off each of the sentences. To make the system appear more flexible, the student is allowed to read the sentences in arbitrary order. The speech recognizer is not constrained to recognize only the sentences displayed, either. Once a sentence is correctly pronounced, the system gives praise and flips the display of the sentence to its parallel counterpart in L1.

When difficulty is encountered, the system provides assistance by synthesizing the specified sentence, as well as displaying an equivalent sentence in L1, and/or other helpful information for pronunciation and understanding. A "give up" button is also available for the student to quit in the middle of a round.

Figure 4-2 illustrates the system diagram. The reading game is a rather simple system in the sense that it does not involve the meaning layer. In initialization of each game round, the content generator generates a list of sentences for the current round as the game sentences. In every game turn, the student's utterance is sent to the performance assessor. The result from the performance assessor goes to the game controller to produce a system act. The language generator is used to generate both a system's response and an updated GUI encoded in HTML, based on the game sentences in the initialization stage, or the system act in each turn.

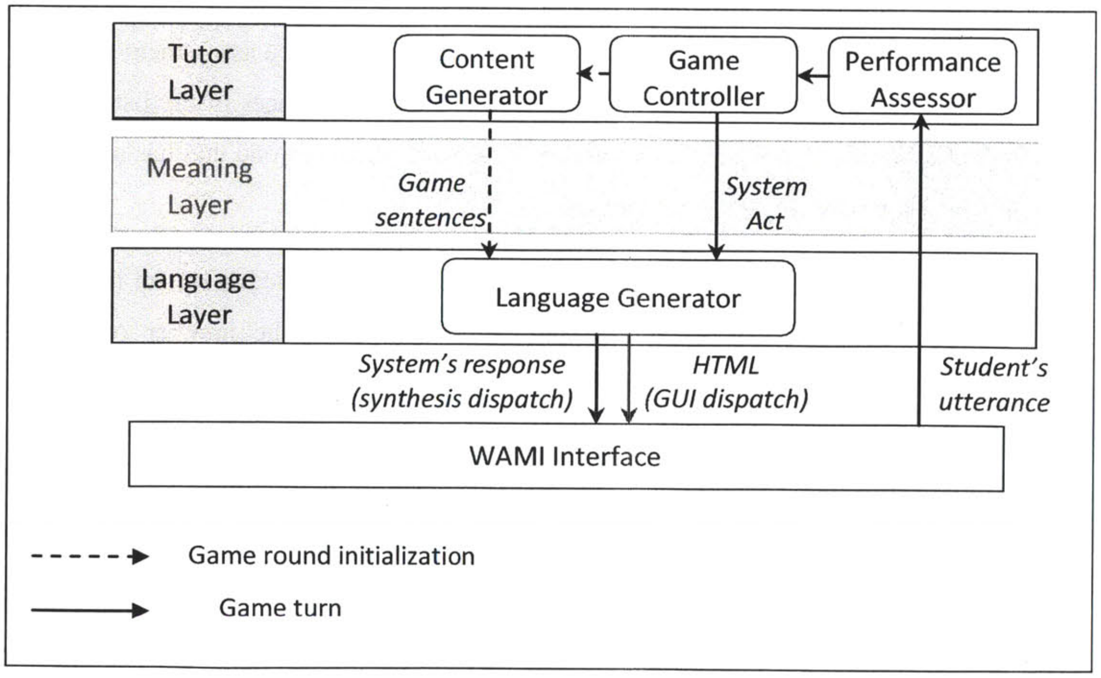


Figure 4-2. System diagram of the reading game.

4.1.2 Contents

The contents of the game are generated from templates described in the previous chapter. There are three choices of the language used to compose the templates: L1, L2, or parallel L1 and L2. Bilingual templates are slightly more complex to prepare than monolingual templates. However, in order to generate both the L2 sentences for the student to read, and the L1 sentences for the student to understand the meaning, using monolingual templates would require an extra translation step to obtain the corresponding sentences in the other language. The advantage, on the other hand, is that the person who prepares the templates can be a monolingual speaker. Particularly, if the templates are in L1, the student himself can potentially be the content provider, and thus the contents can focus on whatever he is interested in.

As a result, we allow two choices of the template language: L1 or parallel L1 and L2. For bilingual templates, the L1 side of the generation outputs is displayed to the student for reading, and the L2 side for the parallel sentences. For templates written in L1 only, we need a method for automatic translation. We will leave the discussion of the translation method in the section on the translation game.

4.1.3 Assessment

The utterance assessment of the reading game is very straightforward: string comparison between the student's utterance and the list of displayed sentences. The utterance assessment outputs one of the following three options: a *match* if one of the unmatched sentences in the list is matched with the student's utterance; a *repeat* if the student's utterance matches one that has already been correctly spoken; or a *none* if no match is found.

When a game round is completed or aborted, a round score is computed to reflect the student's performance in this round. The score is based on three factors: the number of sentences completed, the number of turns the student took, and the number of times the student asked for assistance. If all sentences are completed, each took only one turn, and no assistance is accessed, a score of 100 is given. The exact round score R is calculated using the following Equation 4-1, where n is the number of sentences in each round, t is the number of turns the student took for the sentence, and d is the maximum turns allowed for a sentence in order to obtain some points.

$$R = \frac{100}{n} \sum_{\substack{\text{sentence} \\ \text{completed}}} \frac{\min(0, d - t + 1)}{d} \tag{4-1}$$

The round score *R* translates to an increment of the overall points *ΔP* using Equation 4-2, where *b* is the break-even score, or the pass score. A round score below the break-even score results in a deduction in the overall points.

$$\Delta P = \begin{cases} \frac{100(R - b)}{100 - b} & R \geq b \\ \frac{100(R - b)}{b} & R < b \end{cases} \tag{4-2}$$

The overall points are accumulated across multiple rounds. When enough points have been collected, the student is advanced to the next level, which corresponds naturally to the next lesson in the templates. Equation 4-3 defines the level adjustment *ΔL*, where *m* is the number of root templates in the current lesson, and *n* is the number of sentences in each round. *ΔL* is correlated with *m*, so that lessons with more contents require more rounds to complete. In the case that the overall points are negative, *ΔL* is set to -1, meaning that the system should lower the level by one.

$$\Delta L = \begin{cases} 1 & \Delta P \geq \frac{m}{n} \cdot 100 \\ 0 & 0 \leq \Delta P < \frac{m}{n} \cdot 100 \\ -1 & \Delta P < 0 \end{cases} \tag{4-3}$$

4.1.4 Game control

The game control module takes in the results that the assessment module produces, and outputs the system act. Three system acts are defined.

Correct: when the utterance assessment result is *match*, and there is at least one sentence in the round that has not been matched. The *correct* act produces a praise response, such as "good job" and "well done", and replaces the matched sentence in the user interface with its parallel sentence in L1, so that the student can understand the meaning.

Repeat: when the utterance assessment result is *repeat*. This act produces a prompt to indicate that this sentences has already been completed.

New_round: when all the sentences are matched. The system shows the round score *R*, and adjusts the level according to *ΔL*. A preview of the next round is generated, and the student is asked whether he would like to try another round.

The reading game is a very simple game. It, however, verifies the feasibility and easiness to use the proposed framework and architecture to build speech-enabled language learning systems. It also serves as a basic prototype for more complex systems. As we will see in the next two sections, the design of the translation game and the question-answering game is very similar to that of the reading game. By incorporating more modules on the meaning layer, the reading game can be extended to much more complex games.

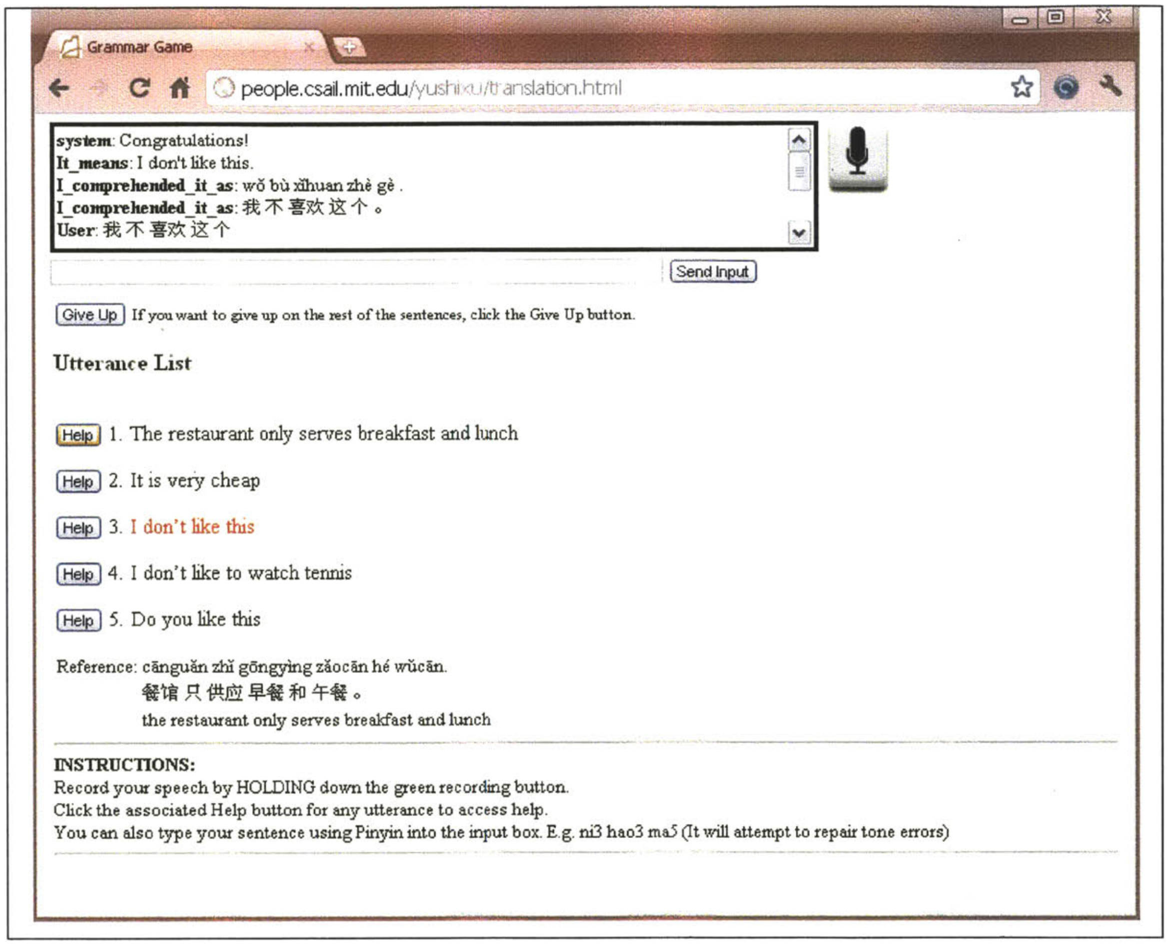
4.2 The translation game

4.2.1 Game overview

When the student is familiarized with the pronunciation, as well as some simple sentence structures and vocabulary in L2, the next activity we provide is translation. In the translation game [**51**], the student's task is to provide sentences in L2 with an equivalent meaning of the L1 sentences displayed on the screen. This task imposes difficulty on the student in that the student has to memorize the corresponding L2 vocabulary, and to come up with a grammatical L2 sentence. It puts challenge on the system, too, for there is no unique correct answer for translation. The utterance assessment is much more complicated than that in the reading game. Furthermore, providing assistance becomes more complicated as well. It is easy to offer assistance in the reading game, because the pronunciations of the words can be looked up in the dictionary easily. In the translation game, however, the student might want to ask about the translation of a portion of a sentence. To prepare a translation for every word and phrase appearing in the sentence list is tedious and almost impossible. Thus, an automatic method is necessary.

Figure 4-3 shows a screenshot of the translation game. The interface is very similar to that of the reading game, except that the game sentences are shown in L1. Since an automatic translation component is necessary anyway, the templates used in this game are monolingual, more specifically, in L1. In practice, the translation game and the reading game can share the same templates, so that there is a smooth transition from the reading game to the translation game for the students.

As in the reading game, the student is allowed to provide translations of the game sentences in any order. The system judges the correctness of the translation against the sentence list. To minimize the impact of mis-recognition. The system first echoes and paraphrases the student's translation to tell the student what it thinks the student said. It then provides a translation back into L1, so that if the translation is incorrect, the student can have a clue of the mistake. If the student's translation is correct, an encouragement is given and the matched sentence is marked. At the end of the round, the system computes a round score and the level adjustment in the same way as it does in the reading game.



system: Congratulations!
It_means: I don't like this.
I_comprehended_it_as: wǒ bù xǐhuan zhè gè .
I_comprehended_it_as: 我 不 喜欢 这 个 。
User: 我 不 喜欢 这 个

If you want to give up on the rest of the sentences, click the Give Up button.

Utterance List

1. The restaurant only serves breakfast and lunch
2. It is very cheap
3. I don't like this
4. I don't like to watch tennis
5. Do you like this

Reference: cānguǎn zhǐ gōngyìng zǎocān hé wǔcān.
餐馆 只 供应 早餐 和 午餐 。
the restaurant only serves breakfast and lunch

INSTRUCTIONS:
Record your speech by HOLDING down the green recording button.
Click the associated Help button for any utterance to access help.
You can also type your sentence using Pinyin into the input box. E.g. ni3 hao3 ma5 (It will attempt to repair tone errors)

Figure 4-3. A screenshot of the translation game in the middle of a round with reference translation for the first sentence. Red marks the sentence that has been completed. A demonstrative video is available at http://people.csail.mit.edu/seneff/scill/translation_game.wmv.

Two kinds of assistance are provided in the translation game. The first is similar to the assistance function in the reading game. The student can click on the "help" button associated with each sentence to see and hear a reference translation. The student can also ask for a translation of a word or a phrase by typing the L1 words into the input box.

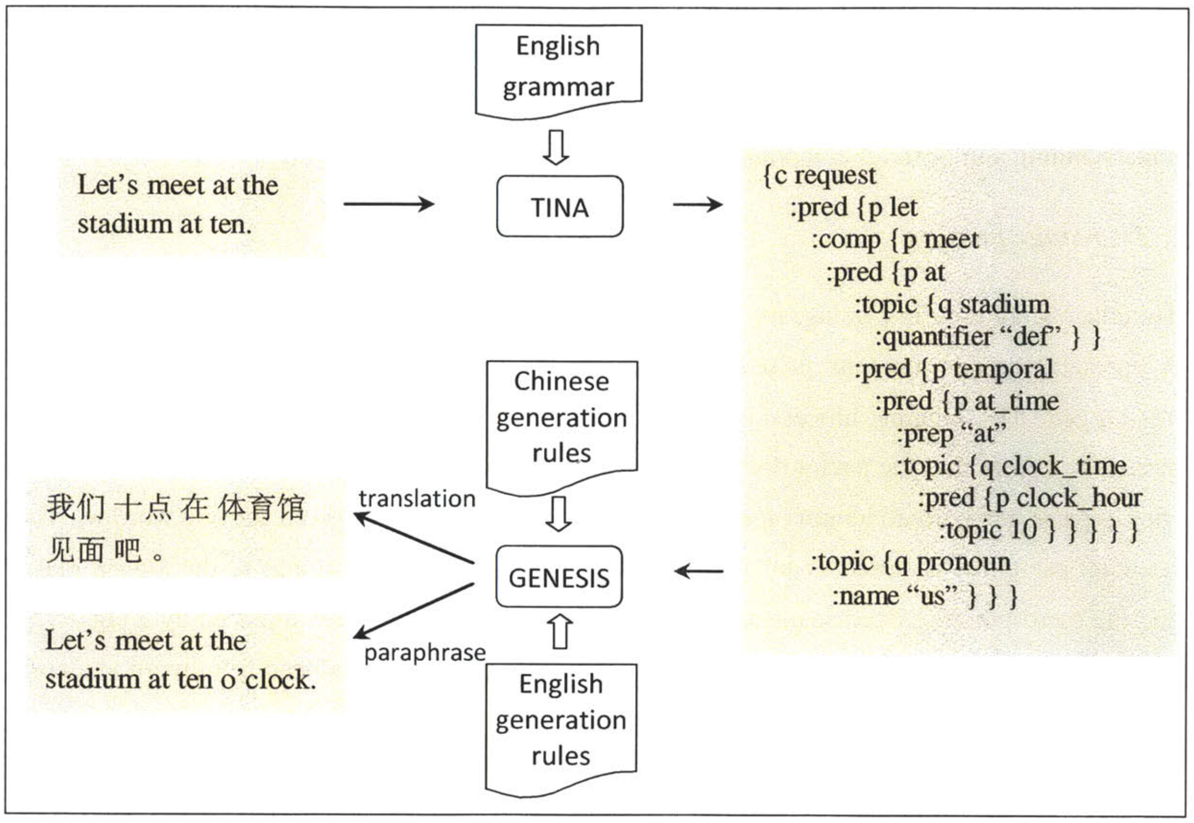
4.2.2 Parse-and-paraphrase paradigm for automatic translation

One of the important capabilities of this system is to automatically generate translations to help the students. Although many statistical machine translation (SMT) methods [52], [53], *etc.*, have been developed and proved quite efficient in the past decades, this domain of interest is different from the domains in which SMT techniques are usually developed. Errors are not tolerable when the automatic translations are used as teaching materials for the student. Besides, for our system is mainly targeted at low-level learners, we can expect that the sentence structures are relatively easy. Thus, we adopt a parse-and-paraphrase paradigm for automatic translation.

The parse-and-paraphrase paradigm has been successfully used in the previous domain-specific version of the translation game [48]. The basic idea is to decompose the translation process into a parsing step and a generation step, interfaced via an interlingua. In our system, this is realized using the language understanding module TINA [54], and the language generation module GENESIS [55]. TINA is a context-free grammar (CFG) parser augmented with non-context-free features such as traces (overt movement restoration), verb-arguments, and other features to improve parsing efficiency on bottom-up languages [56]. Statistical models are trained for the temporal-spatial structure of the parse tree, so as to decide the most likely parse among the ambiguous parses. TINA produces a parse tree and later converts the tree structure into a frame representation, called a parse frame. The parse frame contains a mixture of syntactic and semantic information, and discards the temporal word order in order to be more source-language-independent.

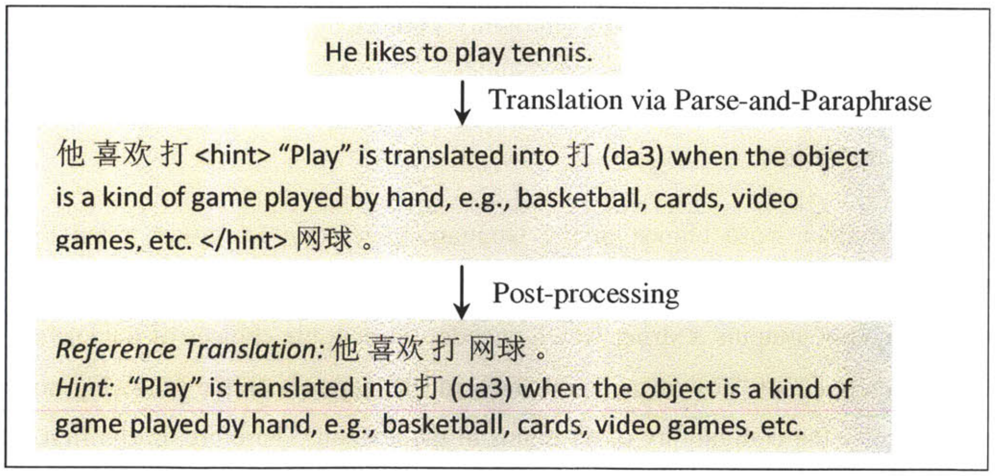
GENESIS, on the other hand, reads in a frame and produces a string according to a set of generation rules and a lexicon. With proper rules and lexicon, GENESIS is capable of producing strings in natural languages, as well as in formal languages such as HTML. Figure 4-4 illustrates the parse-and-paraphrase paradigm. When the input is in English, choosing the English generation rules results in an English paraphrase. If the Chinese generation rules are used instead, the output string becomes a Chinese translation.

With carefully tuned parsing grammar and generation rules, the parse-and-paraphrase paradigm is able to produce translations with good quality. This paradigm also allows an easy generation of grammar points together with the paraphrase/translation. Because the generation is produced by stepping into each constituent (subframe) in a specific order, the grammar points can be encoded into the generation rules of certain constituents, or in the lexicon of a certain vocabulary item. An example is shown in Figure 4-5, where the raw generation output has a grammar point (hint) embedded. The hint, relating to the translation of the verb "play", is separated from the translation in a post-process, and displayed to the student when he asks for assistance.


English grammar
Let's meet at the stadium at ten.
TINA
Chinese generation rules
我们 十点 在 体育馆 见面 吧 。
translation
GENESIS
Let's meet at the stadium at ten o'clock.
paraphrase
English generation rules

```
{c request
   :pred {p let
      :comp {p meet
         :pred {p at
            :topic {q stadium
               :quantifier "def" } }
            :pred {p temporal
               :pred {p at_time
                  :prep "at"
                  :topic {q clock_time
                     :pred {p clock_hour
                        :topic 10 } } } } }
      :topic {q pronoun
         :name "us" } } }
```

Figure 4-4. The parse-and-paraphrase paradigm for paraphrase and translation.


He likes to play tennis.

Translation via Parse-and-Paraphrase

他 喜欢 打 <hint> "Play" is translated into 打 (da3) when the object is a kind of game played by hand, e.g., basketball, cards, video games, etc. </hint> 网球 。

Post-processing

Reference Translation: 他 喜欢 打 网球 。
Hint: "Play" is translated into 打 (da3) when the object is a kind of game played by hand, e.g., basketball, cards, video games, etc.

Figure 4-5. An example of generating translation and grammar points simultaneously.

Although the parse-and-paraphrase paradigm is a good solution for our game, ambiguity in parsing and generation often makes things tricky. When parsing longer sentences, it is harder for the correct parse to appear as the top parse, since the number of ambiguous theories grows exponentially with the length of the sentence. In generation, the choice of words given a meaning is sometimes ambiguous. The dependency on the context oftentimes makes it a difficult problem to handle by manual rules. To better deal with the ambiguities, we extend the language understanding and generation modules into two-stage processes respectively.

Two-stage parsing

To reduce ambiguity in parsing, we reduce the length of the input sentence by a tagging process. A special tagging grammar is used, which defines two classes of root nodes *tagged* and *not_tagged*. The tagging process is then carried out by the regular parser with the tagging grammar, except that the parser does not try to produce a parse for the entire sentence, but only tries to parse a maximum length of word sequences $w_s, \ldots, w_{s+l}$ from a given starting word w_s. The resultant parse tree is stored as an "elemental tree" if its root node r belongs to the *tagged* class, and the corresponding word sequence $w_s, \ldots, w_{s+l}$ in the input sentence is replaced by a tag $<r>$. The parse starts again from the next word w_{s+l+1}. If no parse can be produced from word w_s, word w_s is skipped, and the parser tries again from w_{s+1}.

After the tagging process, the tagged sentence is sent to the parser to parse using the regular grammar. The parse tree is then completed by replacing the leaf nodes which are tags with the corresponding elemental trees. Figure 4-6 illustrates the whole procedure. The nodes that are tagged in the first stage are usually low ambiguity phrases by themselves but can cause great ambiguity when parsed in the whole sentence, such as a complex date and time expression.

Two-stage generation

The context-dependent word choice in the language generation is further aided by using a language model. In the first stage, instead of generating one simple string from an input frame, the generation module outputs a string which encodes all possible choices of the surface words. For example, the choice of "how many" and "how much" depends on the following noun, and it is tedious to specify the countability of all nouns in the lexicon. Therefore, in the first stage, the generation module outputs the following string to include both possibilities.

How (many | much) money is it?

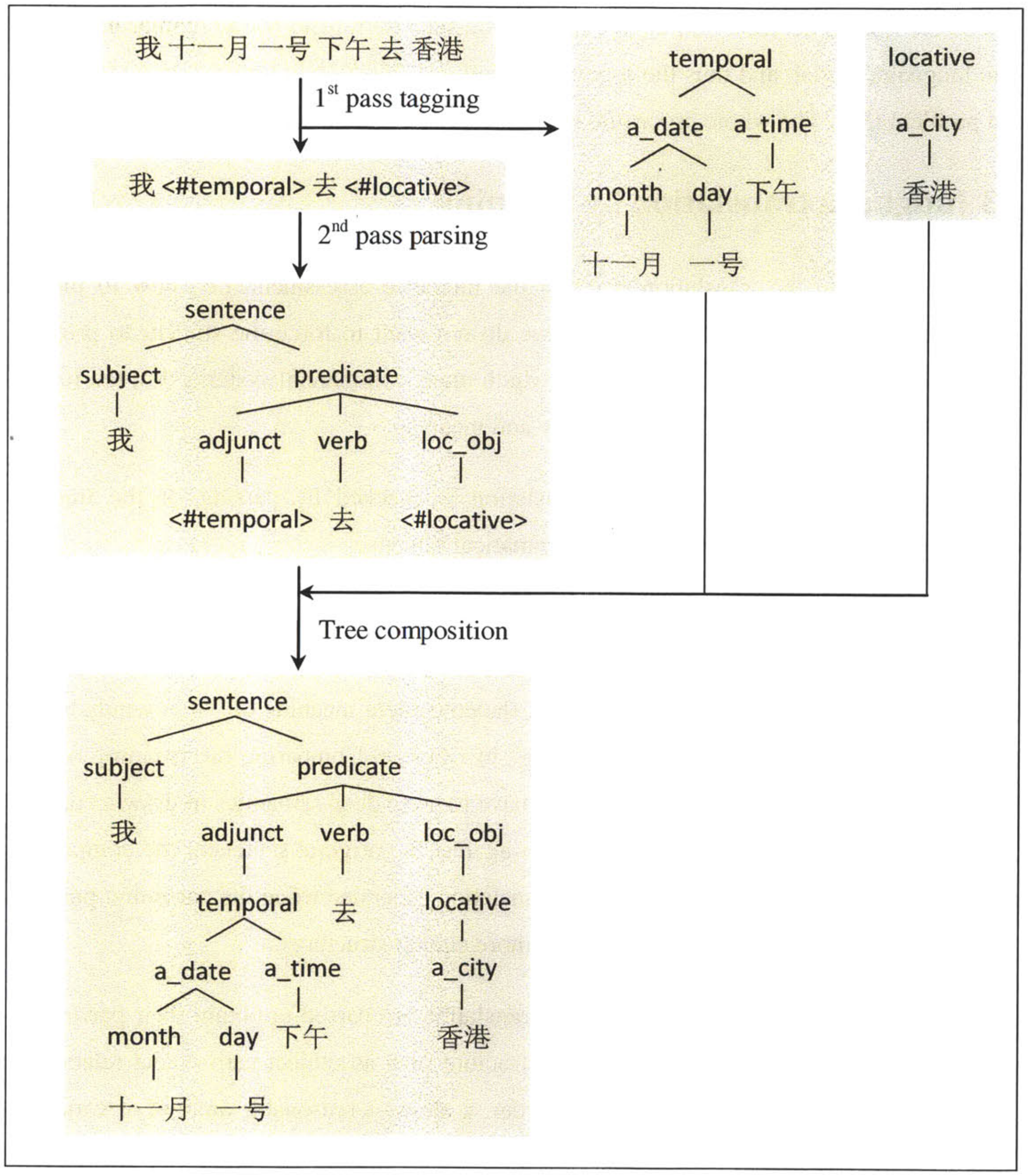


Figure 4-6. Two-stage parsing.

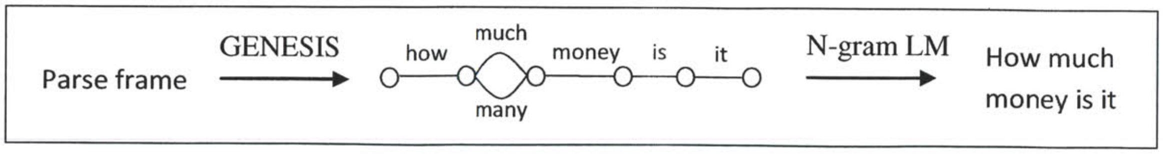


Figure 4-7. Two-stage generation.

In the second stage, the string is converted into a word graph. An *n*-gram language model is then used to rank order the different paths in the word graph. The path with the best score is chosen as the final generation output. We use a finite state transducer (FST) representation of the n-gram language model, and thus the *n*-gram ranking and selection is turned into a standard FST search problem [57]. The whole procedure is illustrated in Figure 4-7.

4.2.3 Judging a translation by meaning

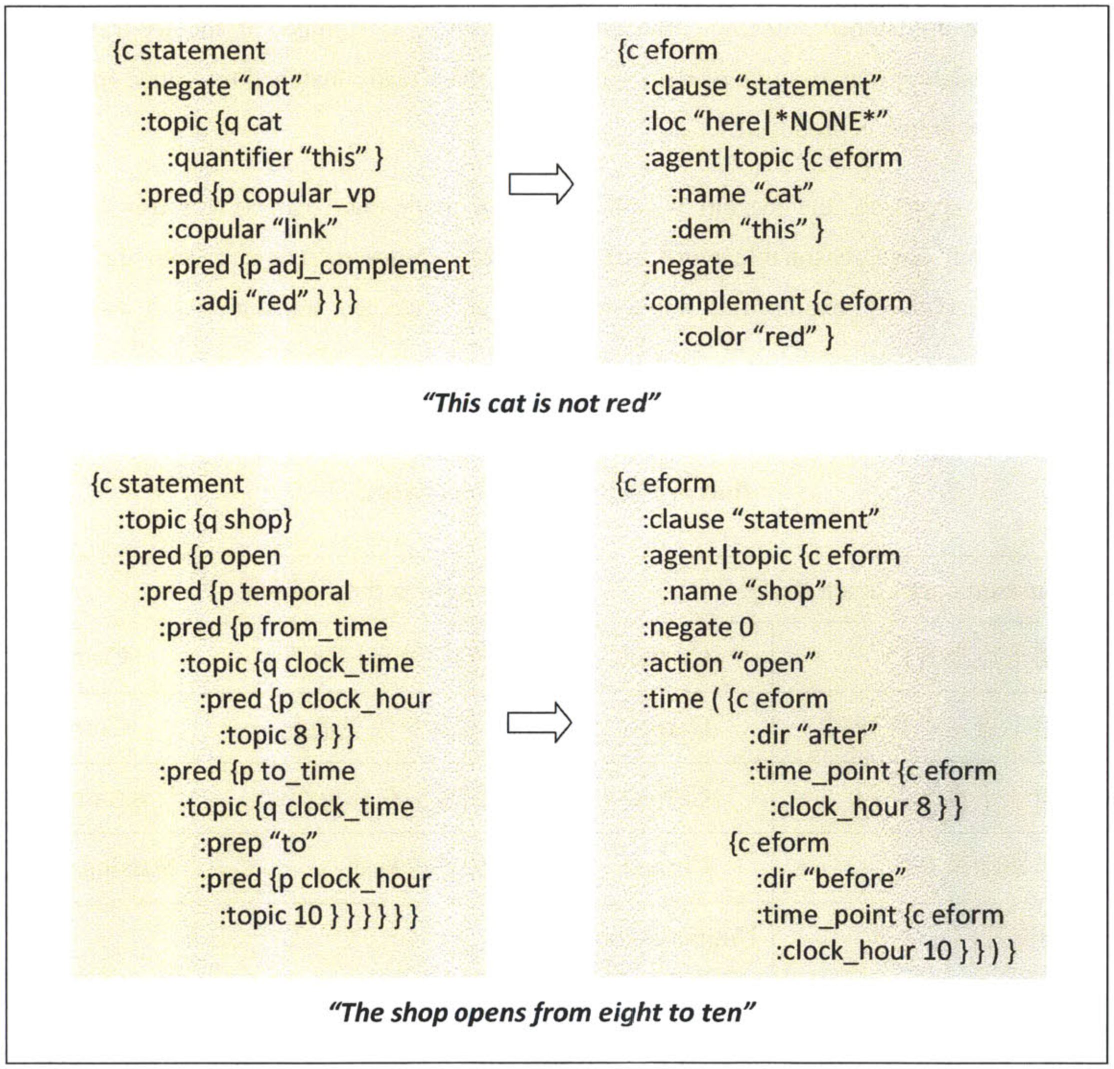
Another challenge in the translation game is the utterance assessment, *i.e.*, how to judge the correctness of the student's translation. For we do not want to force the student to provide an exact translation as the reference translation (which many commercial systems do), the judgment has to be based on two aspects: grammaticality and meaning.

The grammaticality of the student's translation is checked by parsing. If the translation produces a full parse, it is considered as a grammatical sentence.

To compare the meaning with the game sentence, although the sentence in L1 and its translation in L2 should have the same meaning, they may differ in sentence structure due to different language characteristics. For example, the equivalent meaning of "it is windy today" is commonly expressed as "today the wind is big" in Chinese. Comparing the meaning of such a sentence pair is a very challenging task, for we have to go to deep semantics to draw a conclusion. Therefore, instead of comparing the translation against the original sentence, the comparison is done between the translation and a reference translation generated using the parse-and-paraphrase approach, so that the two sentences have much more similar structures.

The student's translation and the reference translation are parsed to obtain their parse frames. The parse frame catches the overall sentence structure such as subject-verb-object relationships, but since it does not keep the temporal word order, it allows a reasonable amount of variation on the surface form. Based on the parse frames, we further analyze the elements and produce a frame that better represents the semantics of the sentence. We refer to this frame as the key-value frames (kv-frames). The keys in the kv-frame can be very domain-dependent, but since the approach we design is meant to be usable for general lesson materials, general semantic roles, such as *agent*, *action*, *patient*, *time*, and *location*, are used as the keys in the kv-frame. Words that belong to the same meaning class are collapsed, such as adjectives "good" and "nice", to provide more robustness. In the situation that the word sense is hard to disambiguate, multiple word senses are kept and separated using a vertical bar to form a disjunction. If an element is omittable, a special value **NONE** is included as one of the disjunctive values.

The generation of the kv-frame from the parse frame is accomplished via GENESIS. GENESIS produces a string representation of the key-value pairs, and a simple algorithm then converts the string into a frame format. Figure 4-8 shows several kv-frames together with their original parse frames.



```
{c statement
   :negate "not"
   :topic {q cat
      :quantifier "this" }
   :pred {p copular_vp
      :copular "link"
      :pred {p adj_complement
         :adj "red" } } }
```

```
{c eform
   :clause "statement"
   :loc "here|*NONE*"
   :agent|topic {c eform
      :name "cat"
      :dem "this" }
   :negate 1
   :complement {c eform
      :color "red" }
```

"This cat is not red"

```
{c statement
   :topic {q shop}
   :pred {p open
      :pred {p temporal
         :pred {p from_time
            :topic {q clock_time
               :pred {p clock_hour
                  :topic 8 } } }
         :pred {p to_time
            :topic {q clock_time
               :prep "to"
               :pred {p clock_hour
                  :topic 10 } } } } } }
```

```
{c eform
   :clause "statement"
   :agent|topic {c eform
      :name "shop" }
   :negate 0
   :action "open"
   :time ( {c eform
              :dir "after"
              :time_point {c eform
                 :clock_hour 8 } }
            {c eform
              :dir "before"
              :time_point {c eform
                 :clock_hour 10 } } ) }
```

"The shop opens from eight to ten"

Figure 4-8. Parse frames and key-value frames.

An algorithm goes through the keys in the kv-frames of the student's translation and the reference translation. A key is matched if one of the disjunctive values in one frame is the same as one of the disjunctive values in the other frame, or if the **NONE** value is contained in its value in one frame, and the key does not appear in the other frame. A substitution error is recorded in the case that the key appears in both kv-frames with different values. An insertion error is recorded in the case that the key appears in the kv-frame of the student's translation but not in the reference kv-frame. A deletion error is recorded in the case that the key exists in the reference kv-frame, but not in the student's kv-frame.

The two kv-frames match if there is no substitution, deletion or insertion error. The utterance assessment produces a *match* if the kv-frame of the student's translation matches any of the kv-frames of the unmatched reference translations. A *repeat* is signaled if the kv-frame of the student's translation matches one of the kv-frames of the already matched reference translations. Otherwise, *no_match* is produced.

Using this approach, the system is able to accept more than one correct translation. The translations can have reasonable word order and word choice differences from the reference translation. Table 4-1 exemplifies some accepted and rejected translations of two English sentences.

Table 4-1. Examples of accepted and rejected translations. Asterisks mark the utterances that are rejected by the system.

The museum opens at ten thirty.		Let's meet at the stadium.	
博物馆十点半开门	Correct	让我们在体育馆碰头	Correct
博物馆十点三十分开门	Correct	咱们在体育馆见面吧	Correct
博物馆于十点半开门	Correct	* 我们碰头在体育馆吧	Ungrammatical
十点半博物馆开门	Correct	* 在体育馆见面	Missing subject
* 博物馆开门十点半	Ungrammatical		

4.2.4 System diagram

With the methods to automatically generate translations and judge translations being introduced, the system diagram of the translation game is shown below in Figure 4-9. Compared to the diagram of the reading game, this game involves modules on the meaning layer. In addition, the modules on the language layer are used more heavily.

In the beginning of every game round, the sentences generated from the templates are processed through the language understanding and language generation modules to produce the reference translations, which are then stored in the session state. After the student has spoken a translation, it is parsed and paraphrased/translated into three things: a paraphrase in L2, a translation in L1, and a string representation of the kv-frame. The kv-frame is then passed to the meaning comparison module to be compared with the reference translations. The performance assessment and game control modules produce a system act according to the comparison result. The system's response and the updated GUI generated based on the system act, if any, are sent to the user via WAMI, following the paraphrase and translation of the student's utterance.

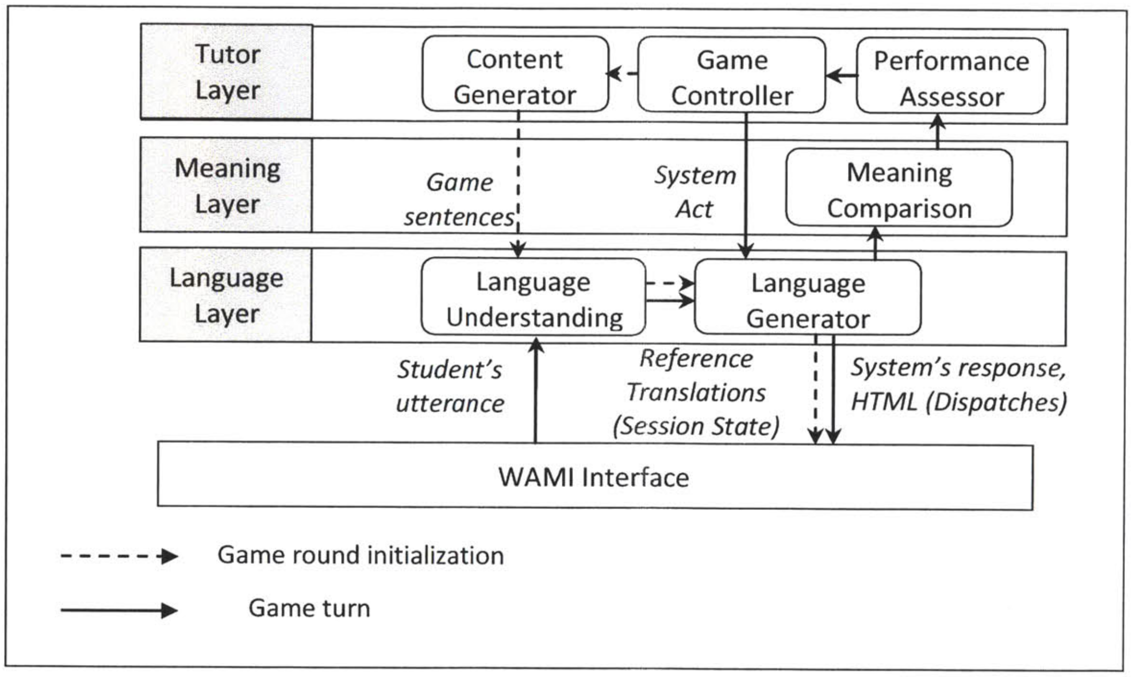
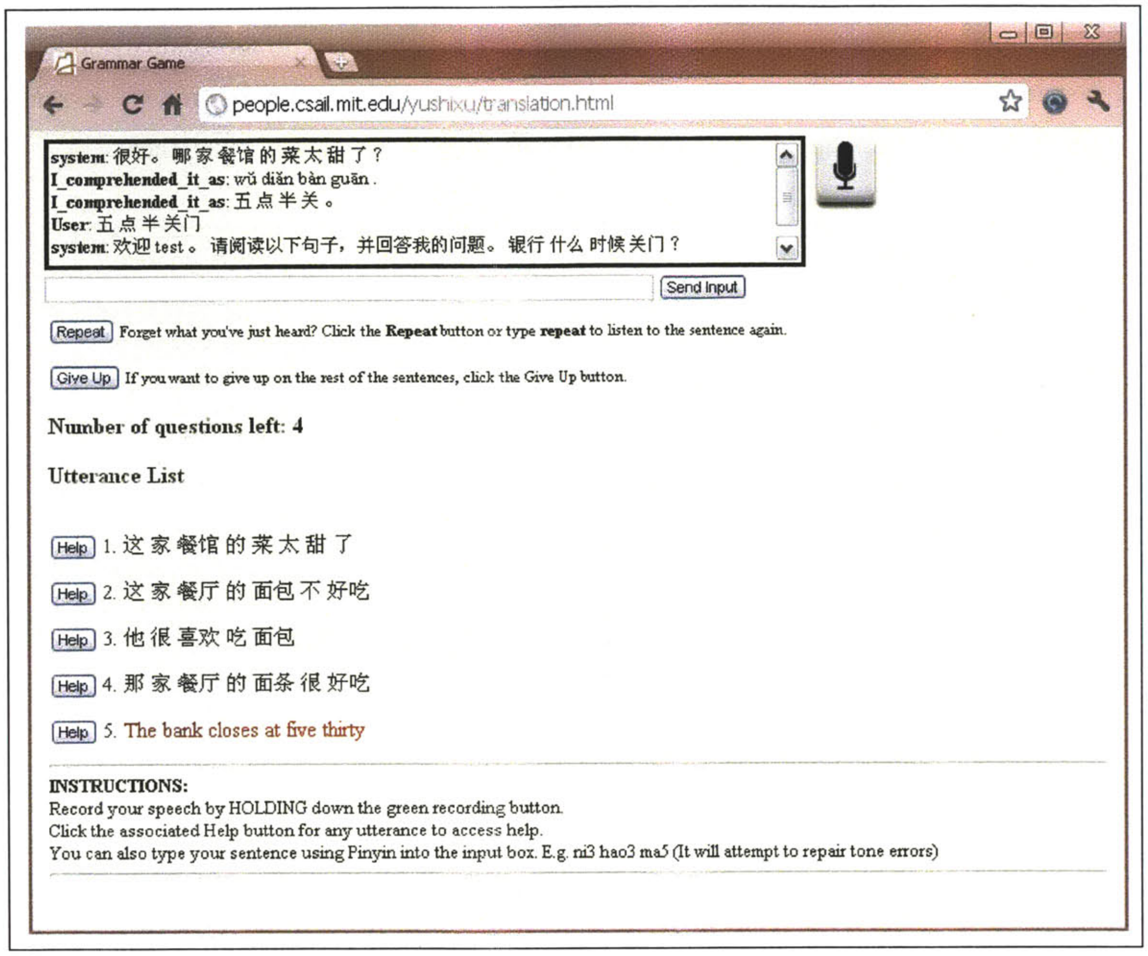


Figure 4-9. System diagram of the translation games.

4.3 The question-answering game

4.3.1 Game overview

The third game [**58**] we developed puts the student in a more interactive L2 environment. In the reading game, the student has seen the sentences in L2 and gotten familiar with the pronunciation. In the translation game, the student has practiced constructing L2 sentences given the L1 counterparts. In the third game, the system displays the game sentences in L2 again, and, in order to force the student to read through and understand the sentences, the system poses spoken questions in L2 based on the sentences, and requires the student to provide answers in L2. We name this game question-answering, which can be viewed as a simplified and spoken version of the traditional reading comprehension exercise. This game exercises a number of language abilities including reading comprehension, listening comprehension, sentence composition and pronunciation, in a complete L2 environment.



system: 很好。哪 家 餐馆 的 菜 太 甜 了？
I_comprehended_it_as: wǔ diǎn bàn guān .
I_comprehended_it_as: 五 点 半 关 。
User: 五 点 半 关门
system: 欢迎 test 。 请阅读以下句子，并回答我的问题。银行 什么 时候 关门？

Forget what you've just heard? Click the **Repeat** button or type **repeat** to listen to the sentence again.

If you want to give up on the rest of the sentences, click the Give Up button.

Number of questions left: 4

Utterance List

1. 这 家 餐馆 的 菜 太 甜 了
2. 这 家 餐厅 的 面包 不 好吃
3. 他 很 喜欢 吃 面包
4. 那 家 餐厅 的 面条 很 好吃
5. The bank closes at five thirty

INSTRUCTIONS:
Record your speech by HOLDING down the green recording button.
Click the associated Help button for any utterance to access help.
You can also type your sentence using Pinyin into the input box. E.g. ni3 hao3 ma5 (It will attempt to repair tone errors)

Figure 4-10. A screenshot of the question-answering game in the middle of a round. The student has already answered one question correctly. Red marks the corresponding statement of that question. The Chinese characters in the dialogue history box translate into (from bottom up) "System: Welcome. Please read the following statements and answer my question. When does the bank close? User: (It) closes at five thirty. System: Very good. Which restaurant's food is too sweet?" A demonstrative video of the system is available at http://people.csail.mit.edu/seneff/scill/Q&A.wmv.

Figure 4-10 shows a screenshot of the question-answering game in the middle of a game round. Again, this is a very similar interface with the previous two games. For each round, the system generates a list of game sentences in L2. To simplify the design of the system, the templates used in this game are bilingual, to eliminate the necessity of the automatic translation when providing assistance. The templates are also written in a way such that all the instantiated sentences are statements, for which questions can be posed (and thus the game sentences are referred to as game statements from now on).

The system generates one question for each game statement. Unlike the previous two games in which the students takes control of the order of game sentences to practice, in the question-answering game, the control is in the system's hand. The system poses the questions in a random order. The student needs to read all the statements displayed on the screen to find out the one that corresponds to the question, and provide a spoken answer. The answer can be in any legal form: abbreviated, complete, or any other legal form between the two. The system then judges whether the answer is correct both syntactically and semantically. If it is correct, the system highlights the corresponding statement of the last question, and moves on to the next one. If the answer is partially correct, *i.e.*, the student provided some but not all of the expected information, the system composes a follow-up question to guide the student towards an expected answer. Otherwise, the student is given two more chances to try. If he fails all three times, the system speaks the answer, and moves on to the next question. Unlike the reading game and the translation game, in this game only the statement corresponding to the last question is highlighted and shown in L1, to prevent the student from cheating by only looking at the unhighlighted statements for the later questions in a round.

The assistance function in this game is inherited from the previous games. A "help" button associates with each game statement to show the translation into L1.

As we can see from the explanation of the game, there are two critical problems to solve to make the game happen: generating questions from given statements, and judging the correctness of an answer given the context. The next two subsections explain our approaches.

4.3.2 Question generation using frame transformation

At first glance, converting a statement into a question seems to be as easy as replacing a word or a phrase in the statement with a corresponding interrogative word. But thinking more carefully, in a language that has overt wh-movement such as English, the interrogative word needs to be moved to the front, and an appropriate auxiliary word needs to be added if there is not any in the original statement. In Mandarin Chinese, although there is no such kind of movement, word order and word choices for a statement and a question are not always the same. For instance, examine the different positions of the character "了" (a tense auxiliary word) in the following three sentences.

他买了书。(He bought a book.)
他买了什么？ (What did he buy?)
他买书了吗？(Did he buy a book?)

Furthermore, the choice of the interrogative word not only depends on the word or phrase being questioned, but also depends on the syntactic/semantic role of the word/phrase in the sentence. As in the following example, when questioned on the word "hotel", "what" and "where" are chosen for the two sentences respectively.

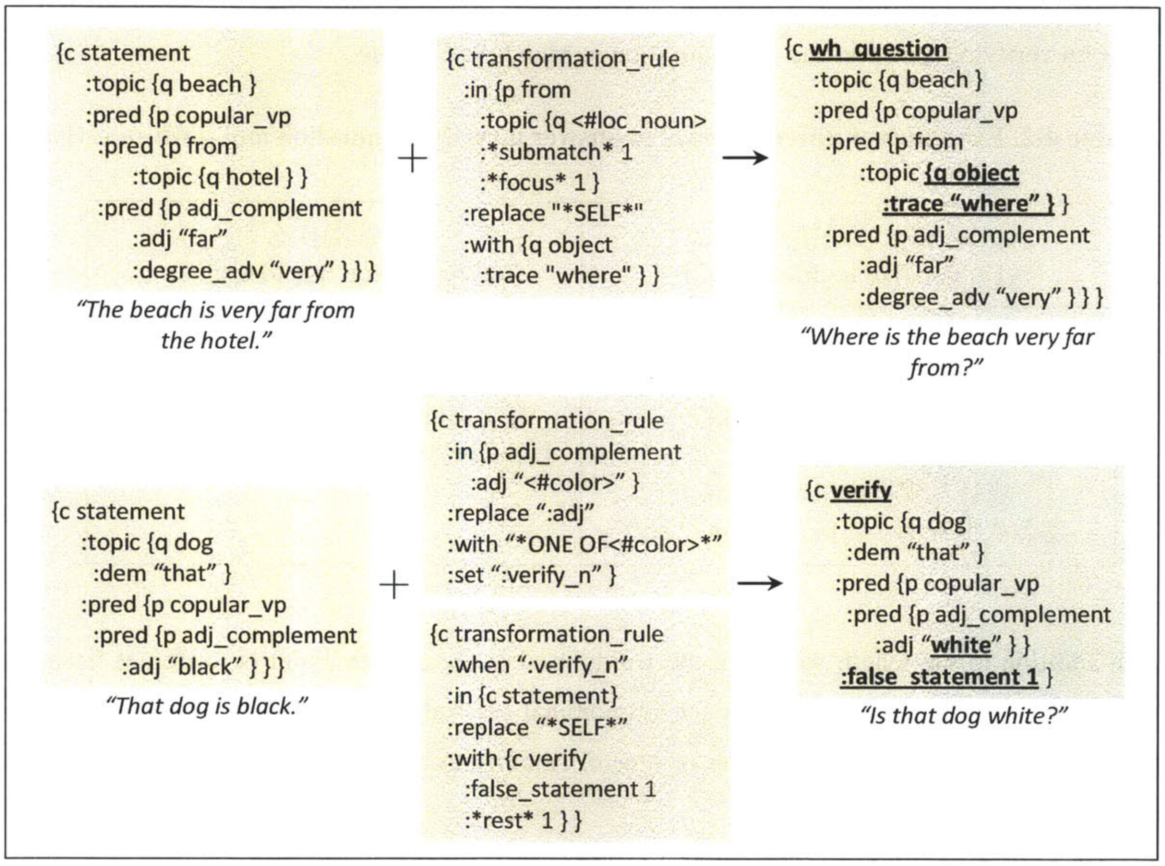
There is a hotel on the corner. → What is there on the corner?
It is near to a hotel. → Where is it near to?

Therefore, we need an approach that is more than string surgery.

The solution turns out to be rather simple. As indicated in the above sections, the parse frame that the language understanding module TINA produces preserves the syntactic hierarchy of the sentences, as well as containing certain explicit semantic elements. In the translation game, generation rules have already been created to turn a parse frame into a string in the original language to perform a paraphrase. Generating a question under these circumstances means that we only need to perform a parse frame modification to change the type of the clause, and for wh-questions, to replace certain elements with the interrogative form. After that, the generation rules will handle the word order and word choice issues automatically. This is also a language independent solution, because the parse frames are, for the most part, source language independent. Thus, whichever language the statement is in, the same types of parse frame modification can be applied.

The modification of the parse frame is realized by applying a set of formal rules, called "transformation rules." Each rule has three basic clauses, which describe the conditions under which the rule should be triggered, the part to be transformed, and the result after transformation. Wildcard values like *ANY*, *NONE*, *SELF*, *etc.*, are adopted in the syntax to make the rules simple to write but powerful to express all kinds of transformations. [**59**] provides a detailed description of the syntax of the transformation rules. The transformation rules can have a simple function such as changing the value of a key, or may describe a complex operation, especially when rules are combined in sequence. They also support some randomness, allowing alternative outputs depending on a randomly generated outcome.

Two examples using this approach are given in Figure 4-11. In the first example, the topic "hotel" under predicate "from" is changed into a generic object with a trace (the nomenclature derives from English wh-movement) "where". Then the name of the clause is changed to "wh_question" by a default setting. The macro "<#loc_noun>" specifies a set of location nouns to share the same transformation rules. The result of the transformation produces a question "where is the beach very far from?" from the original statement "the beach is very far from the hotel."



```
{c statement
   :topic {q beach }
   :pred {p copular_vp
     :pred {p from
        :topic {q hotel } }
     :pred {p adj_complement
        :adj "far"
        :degree_adv "very" } } }
```

"The beach is very far from the hotel."

+

```
{c transformation_rule
  :in {p from
     :topic {q <#loc_noun>
     :*submatch* 1
     :*focus* 1 }
  :replace "*SELF*"
  :with {q object
     :trace "where" } }
```

→

```
{c wh_question
   :topic {q beach }
   :pred {p copular_vp
     :pred {p from
        :topic {q object
          :trace "where" } }
     :pred {p adj_complement
        :adj "far"
        :degree_adv "very" } } }
```

"Where is the beach very far from?"

```
{c statement
   :topic {q dog
     :dem "that" }
   :pred {p copular_vp
     :pred {p adj_complement
        :adj "black" } } }
```

"That dog is black."

+

```
{c transformation_rule
   :in {p adj_complement
     :adj "<#color>" }
   :replace ":adj"
   :with "*ONE OF<#color>*"
   :set ":verify_n" }
```

```
{c transformation_rule
   :when ":verify_n"
   :in {c statement}
   :replace "*SELF*"
   :with {c verify
     :false_statement 1
     :*rest* 1 } }
```

→

```
{c verify
   :topic {q dog
     :dem "that" }
   :pred {p copular_vp
     :pred {p adj_complement
       :adj "white" } }
   :false_statement 1 }
```

"Is that dog white?"

Figure 4-11. Two examples of using the transformation rules to rewrite the parse frames and generate questions. Underlined text marks the difference from the original frame after the transformation.

The second example exemplifies the generation of a "false" verifying question, with a special key *:false_statement* added to be distinguished from a "true" verifying question. The frame for "is that dog white?" is obtained from the original frame representing "that dog is black" by changing the name of the top frame and replacing the color with another color defined in the macro "<#color>." Note that randomness is involved in this transformation. The value of the key *:adj* can be any color on the macro list.

4.3.3 Answer judgment via simplified context resolution

There are many correct ways to answer a question given a statement. The length of the answer can vary from a one-word answer, to the full length of the statement. This is especially true for languages that have a freer grammar, such as Chinese. Table 4-2 summarizes different ways to answer a verifying question and a wh-question in Mandarin Chinese.

Table 4-2. Examples of different ways to answer a verifying question and a wh-question.

你喜欢喝啤酒吗？ (Do you like to drink beer?)	你喜欢喝什么？ (What do you like to drink?)
是的。(yes) 喜欢。(like) 喜欢喝。(like to drink) 我喜欢。(I like) 我喜欢喝。(I like to drink) 我喜欢喝啤酒。(I like to drink beer)	啤酒。(beer) 喝啤酒。(drink beer) 喜欢喝啤酒。(like to drink beer) 我喜欢喝啤酒。(I like to drink beer)

In addition to the length variation, the words used in the answer might be different from the statement. One obvious example is the pronominal referral, where pronouns are used in the answer instead of the explicit person or object. Another example shown below has to do with negation and antonym.

Statement: The hotel is close to the beach.
Is the hotel far from the beach?
No, not far.

To properly judge the correctness of the answer, both the length variation and the word variation need to be taken into consideration. The method we developed divides the judgment into two steps. First, analogous to context resolution, the student's answer is augmented with the information in the question to form a complete answer. Then, the complete answer is compared with the original statement to output the utterance assessment result.

4.3.3.1 Answer augmentation

The answer augmentation works as a simplified context resolution algorithm. The algorithm is applied on the kv-frames, and does not depend on any domain-specific knowledge. It is treated as a discourse phenomenon, *i.e.*, later information (the student's answer) overwrites the earlier information (the question). Since the kv-frames are still hierarchical, and the question and answer kv-frames are very likely to have different sizes and depths (the answer kv-frame is usually smaller and shallower), the two kv-frames need to be aligned before overwriting information from one to the other.

To align two frames *s* and *t* means to find the correspondence of every subframes in *s* with the subframes in *t*. Once the subframe correspondences are established, the alignment of the elements within one aligned subframe can be easily figured out by the name of the keys. For general frame alignment, similarity of the elements in the frames is a good feature. In this particular case, however, there is a stronger cue of how the question kv-frame and the answer kv-frame should be aligned. We observe that, for wh-questions, the question kv-frame must contain one key which has the special coded value "*question*", and in the kv-frame that derives from any utterance that tries to answer the question, the same key with a concrete value must appear. We call this key an anchor key, *i.e.*, after the two kv-frames are aligned, the anchor keys from the two frames should be overlapping.

For verifying questions, there is no such key with the value "*question*", so the alignment is done using the similarity of the frame content alone, while both the anchor keys and the similarity are used in the case of a wh-question. We define a set of keys called "useful keys" which exclude less important keys for the alignment, to calculate the similarity of frame content. Equation 4-4 defines the matchedness score *m* for a given question kv-frame *Q* and an answer kv-frame *A* based on the similarity of the content *sim(·)* and $n_{\overline{aa}}$, the number of unaligned anchor keys in *Q*. For verifying questions, $n_{\overline{aa}}$ is always zero. The similarity is computed using Equations 4-5 and 4-6, where n_Q is the number of useful keys in *Q*, *k* is a key, and *v* is the value of the key *k* in the frame.

$$m(Q,A) = \frac{1}{(n_{\overline{aa}}+1)} sim(Q,A) \tag{4-4}$$

$$sim(Q,A) = \frac{1}{n_Q} \sum_{k \in Q \cap A} s(Q,A,k) \tag{4-5}$$

$$s(Q,A,k) = \begin{cases} 0 & k \notin Q \; or \; k \notin A \\ 1 & k \in A, v_Q = \text{"} * question * \text{"} \\ \dfrac{1+m(v_A,v_Q)}{2} & v_Q, v_A \; are \; frames \\ 1 & v_Q = v_A, v_Q \; and \; v_A \; not \; frames \\ 0 & otherwise \end{cases} \tag{4-6}$$

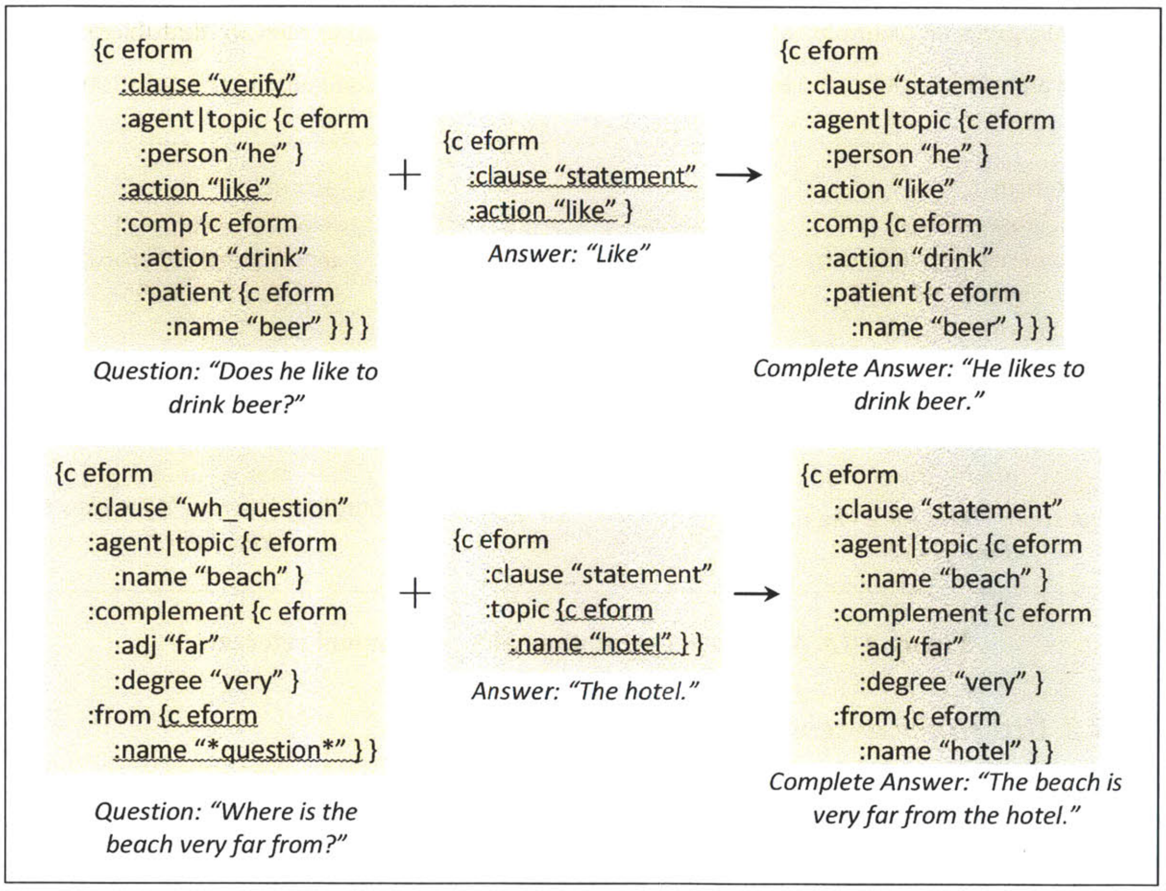
Using the matchedness score m, the best alignment can be found. Assuming that the size and depth of the answer kv-frame *A* are always smaller or equal to those of the question kv-frame *Q*, the algorithm to find the best subframe in *Q* to be aligned with *A* goes as follows (except one-bit yes-no answer for verifying questions, which will be explained later).

```
1. Compute $m_{top} = m(Q, A)$
2. For every subframe $s_i$ in Q, find the best subframe in $s_i$ to be aligned with A, with score $m_i$.
3. Find the subframe sub with the best score.
4. If $m_{top} > m_{sub}$, return Q with score $m_{top}$; otherwise return $s_{sub}$ with score $m_{sub}$.
```

Once the question kv-frame and the answer kv-frame are aligned, a complete answer kv-frame is obtained by overwriting the elements in the question kv-frame with the corresponding elements in the answer kv-frame, and adding new keys from the answer kv-frame that do not exist in the question kv-frame. Two examples are shown in Figure 4-12, with the aligned parts of the frames underlined.

The first example is a simple case for a verifying question "他喜欢喝啤酒吗？" (does he like to drink beer). The kv-frame of the answer "喜欢" (like) only contains two keys: the clause type, and the action "like". The alignment result aligns the answer kv-frame with the top frame of the question kv-frame. After overwriting the elements, the resulting complete answer kv-frame, except for the type of the clause, is exactly the same as the question kv-frame, which represents the meaning of "he likes to drink beer."

In the second example, the question is a wh-question. The kv-frame of the question contains a key with the value "*question*", which serves as the anchor key. The same key can be found in the answer kv-frame, and as a result, the two subframes containing the anchor key are aligned together. Notice that in this example, the top level frame of the answer frame is not aligned to any part of the question kv-frame, which is only allowed when the top level of the answer kv-frame only contains one useful key (*:clause* is not a useful key). This rule is made especially for short answers of wh-questions, because in short answers, the word or the phrase, especially the noun phrase loses its syntactic/semantic role in the question, which usually results in a different top-level key from expected (*:topic* in the example, rather than *:from*). Therefore, it is allowed to skip the top-level frame, and directly align the second-level subframe.



```
{c eform
   :clause "verify"
   :agent|topic {c eform
     :person "he" }
   :action "like"
   :comp {c eform
     :action "drink"
     :patient {c eform
        :name "beer" } } }
```

Question: "Does he like to drink beer?"

+

```
{c eform
   :clause "statement"
   :action "like" }
```

Answer: "Like"

→

```
{c eform
   :clause "statement"
   :agent|topic {c eform
     :person "he" }
   :action "like"
   :comp {c eform
     :action "drink"
     :patient {c eform
        :name "beer" } } }
```

Complete Answer: "He likes to drink beer."

```
{c eform
   :clause "wh_question"
   :agent|topic {c eform
     :name "beach" }
   :complement {c eform
     :adj "far"
     :degree "very" }
   :from {c eform
     :name "*question*" } }
```

Question: "Where is the beach very far from?"

+

```
{c eform
   :clause "statement"
   :topic {c eform
     :name "hotel" } }
```

Answer: "The hotel."

→

```
{c eform
   :clause "statement"
   :agent|topic {c eform
     :name "beach" }
   :complement {c eform
     :adj "far"
     :degree "very" }
   :from {c eform
     :name "hotel" } }
```

Complete Answer: "The beach is very far from the hotel."

Figure 4-12. Examples of obtaining complete answer kv-frames from short answer kv-frames and question kv-frames.

The following paragraphs discuss some handling methods for several special issues.

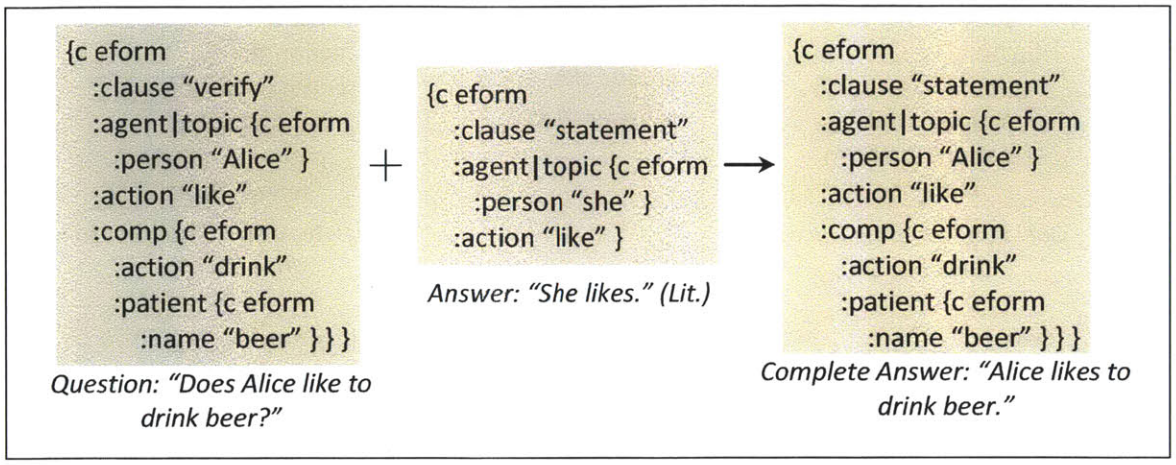
Augmentation for one-bit yes/no answer

If the answer to a verifying question only contains a "yes" or a "no", the augmentation is simple. The question kv-frame is used as the complete answer kv-frame, changing the clause type from a verifier to a statement. If the answer is "no", a negation is added to the top level of the frame.

Pronominal referral

The pronominal referral problem is solved by assigning a special transparent property to the pronouns. When overwriting elements in the question kv-frame with those in the answer kv-frame, the pronouns in the answer kv-frame are transparent, *i.e.*, they will not overwrite the values of corresponding keys in the question kv-frame. However, if there is no corresponding key in the question kv-frame to overwrite, the pronouns will survive in the final complete answer kv-frame.

Figure 4-13 gives an example, where in the answer a pronoun is used to refer to the subject. In the complete answer kv-frame, the agent is resolved to the original person name in the question.



```
{c eform
  :clause "verify"
  :agent|topic {c eform
    :person "Alice" }
  :action "like"
  :comp {c eform
    :action "drink"
    :patient {c eform
      :name "beer" }}}
```

Question: "Does Alice like to drink beer?"
+

```
{c eform
  :clause "statement"
  :agent|topic {c eform
    :person "she" }
  :action "like" }
```

Answer: "She likes." (Lit.)
→

```
{c eform
  :clause "statement"
  :agent|topic {c eform
    :person "Alice" }
  :action "like"
  :comp {c eform
    :action "drink"
    :patient {c eform
      :name "beer" }}}
```

Complete Answer: "Alice likes to drink beer."

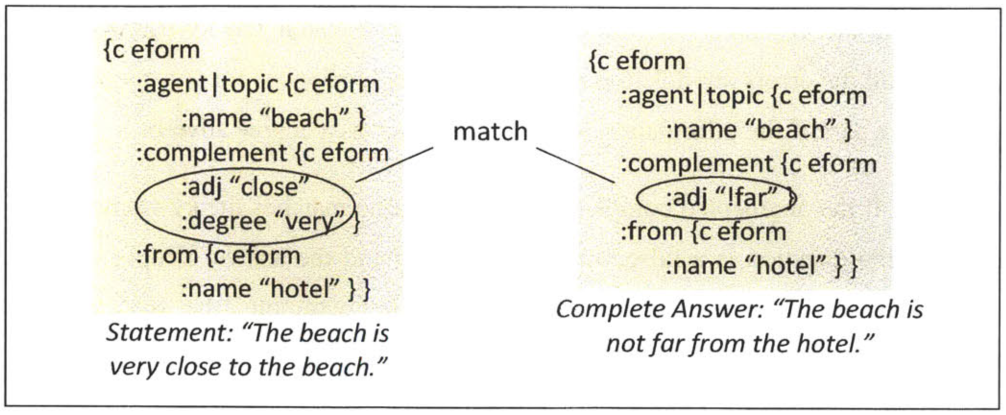
Figure 4-13. Answer augmentation with pronominal referral.

4.3.3.2 Utterance assessment and game control

After the kv-frame of the student's answer has been augmented into a complete answer kv-frame, it is compared to the kv-frame of the original statement. The approach is a variant of the one we have introduced in Section 0 for the translation game, with more flexibility to interpret negations, as well as including an additional check for self-contradictory answers to verifying questions.

Negations

Consider the example we have shown at the beginning of this section. When the student is given the statement "the hotel is close to the beach", and then asked "is the hotel far from the beach," answering "not far" is natural and correct. To accept this answer, the system interprets "not far" and "close" to have the same meaning by propagating the negation at the clause level to the lower levels, as exemplified in Figure 4-14. The exclamation mark indicates the negation of the value. In the comparison, "!far" and "close" are considered to be a match, and the degree "very" in the statement is ignored because of the negation.



```
{c eform
   :agent|topic {c eform
       :name "beach" }
   :complement {c eform
       :adj "close"
       :degree "very" }
   :from {c eform
       :name "hotel" } }
```

Statement: "The beach is very close to the beach."

```
{c eform
   :agent|topic {c eform
       :name "beach" }
   :complement {c eform
       :adj "!far" }
   :from {c eform
       :name "hotel" } }
```

Complete Answer: "The beach is not far from the hotel."

Figure 4-14. Handling negations in the judgment.

Self-contradictory answers to verifying questions

There are three possibilities for an answer to a verifying question. It can be a short "yes" or "no," a statement, or the combination of the two. In the case of the combination answers, there is the possibility that the answer by itself is a contradiction. For example, the student is asked "is that dog white," and he answers "yes, it is black." This is not an acceptable answer, since the two parts of the utterance lead to contradictory meanings. The system rejects this kind of self-contradictory answers by splitting the utterance in two, one part containing only the word "yes" or "no", the other containing the remaining statement. The split two pieces are then augmented independently, and a meaning comparison is carried out between the two augmented kv-frames. If they do not match each other, a contradiction flag is set.

Pronouns "you" and "I"

There is a switch of the pronoun "you" and "I" in the question/statement and the answer. If the question asks about "you", the student should answer with "I" with the proper case, and vice versa. For correct comparison, the pronouns "you" and "I" in the statement are replaced by each other, to match the student's angle.

Utterance assessment result and game control

Four result types are defined for the utterance assessment in the question-answering game.

No_match: if the augmented complete answer kv-frame does not match the kv-frame of the original statement, or the answer is a self-contradiction.

Incomplete: if the complete answer kv-frame does not match the kv-frame of the original statement, and all the errors are deletion errors. This usually happens for wh-questions, where the student left out a part of the information in the answer such as a degree adverb.

Match_false: if the augmented complete answer kv-frame matches the kv-frame of the original statement, but the question is a false verifying question and the answer simply denies the false information without providing the true information. For example, the student is given the statement "the dog is white" and asked "is the dog black?" The answer "no, it is not black" is considered as incomplete. The expected answer would be "no, it is white."

Match: otherwise.

The game control module takes in the result of the utterance assessment, and produces one of the following five system acts. More acts are defined for this game than the previous two games, since the utterance assessment module produces more detailed results.

Correct: the utterance assessment result is *match*, and there is at least one question left for the round. The system marks the corresponding statement, and goes on to the next question.

More_precise: the utterance assessment result is incomplete, and the student still has more chances to answer the question. The system prompts the student to give a more precise answer.

Follow_up: the utterance assessment is *match_false*. In this case, the system generates a follow-up question to help the student provide a complete answer. The follow-up question is essentially the difference between the kv-frame of the statement and the augmented complete kv-frame of the answer, as shown in Figure 4-15. It is first produced in the kv-frame representation, and later paraphrased into a natural sentence via the language generation module. This is an interesting situation, for the follow-up question extends the question-answering into a multi-turn interaction, and makes the exercise closer to a dialogue.

Try_again: the utterance assessment is *no_match*, and the student has not used up all the three chances. The system repeats the question and asks the student to try again.

No_more_chance: the utterance assessment is *incomplete* or *no_match*, the student has used up all the three chances, and there is at least one question left for the round. In this case, the system speaks the correct answer, and moves on to the next question.

New_round: the system is ready to move on to the next question either because the utterance assessment is *match*, or because the student has used up all the three chances, but there is no question left for the round. The system computes the student's round score and the level adjustment, and shows a preview of the next round.

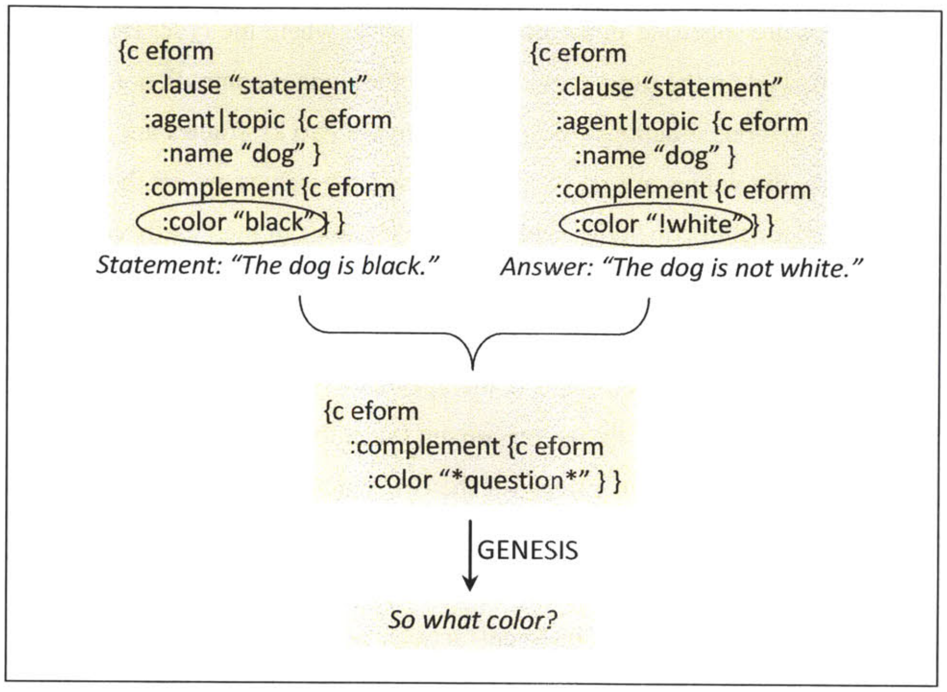


Figure 4-15. Generation of a follow-up question.

4.3.4 Contradiction detection

So far, it seems that every problem has been solved to build the question-answering game. Nevertheless, as soon as we started to test the system using some toy templates, we noticed one piece missing. Remember that the game statements are generated from the templates in a random fashion. For small lesson templates, it is highly probably to generate two statements from the same template and containing contradictory information. For example,

Statement: The dog is white.
The dog is black.
Question: What color is the dog?

This results in a confusing situation for the student, for the student is not able to tell which statement he is supposed to use as the reference. To avoid this kind of situation, a detection algorithm is designed to reject a list of game statements which contains contradiction.

The detection applies on the kv-frames of the game statements. A set of ordered "dominant" keys is defined. The keys are dominant if different values can be used to distinguish different events among which contradiction should not happen. For example, if the agents are different, the two statements should not contradict because they are talking about different events.

The dominant keys are specified in an ordered sequence, where the order represents the ranks. Multiple keys are allowed on the same rank. The algorithm judges contradiction rank by rank. For a given rank, there are three situations: the values from the two kv-frames are different for all of the keys on the rank, the values are partly different, and the values are all the same. In the "all different" case, the algorithm concludes with no contradiction. In the "partly different" case, the algorithm signals a contradiction. In the last case, the decision is deferred to the lower ranks. To be more intuitive, in practice, the key *:agent* is placed on the highest rank, followed by keys *:action* and *:time* on the second rank. If the agents of the two statements are different, the two statements do not contradict. If the agents are the same, the algorithm checks the next rank. If both *:action* and *:time* have different values for the two statements, which means the same agent is doing different things at different times, the algorithm concludes with no contradiction. But if *:action* has different values but *:time* has the same value, a contradiction is signaled because the same agent should not perform two different actions at the same time. The other situation where *:action* has the same value but the *:time* has different values does not always produce a contradiction in the real world, but for some actions, it does result in a contradiction. For example, "the shop opens at nine" and "the shop opens at ten" are a contradictory pair. Thus, the algorithm signals a contradiction whenever a portion of the keys on the same rank have different values in the two statements.

The algorithm goes through all the ranks of the dominant keys. Keys that do not appear in either of the kv-frames are skipped. If no conclusion has been made after all the dominant keys have been examined, the algorithm checks the non-dominant keys. Any difference in the values of the non-dominant keys results in a contradiction.

The dominant key list is specified manually. However, the list is rather simple (the final list only consists of five ranks with altogether less than ten keys), and we consider the list to be generic to any domain.

4.3.5 System diagram

Now that we have introduced all the important modules for the question-answering game, Figure 4-16 presents the system diagram of the game.

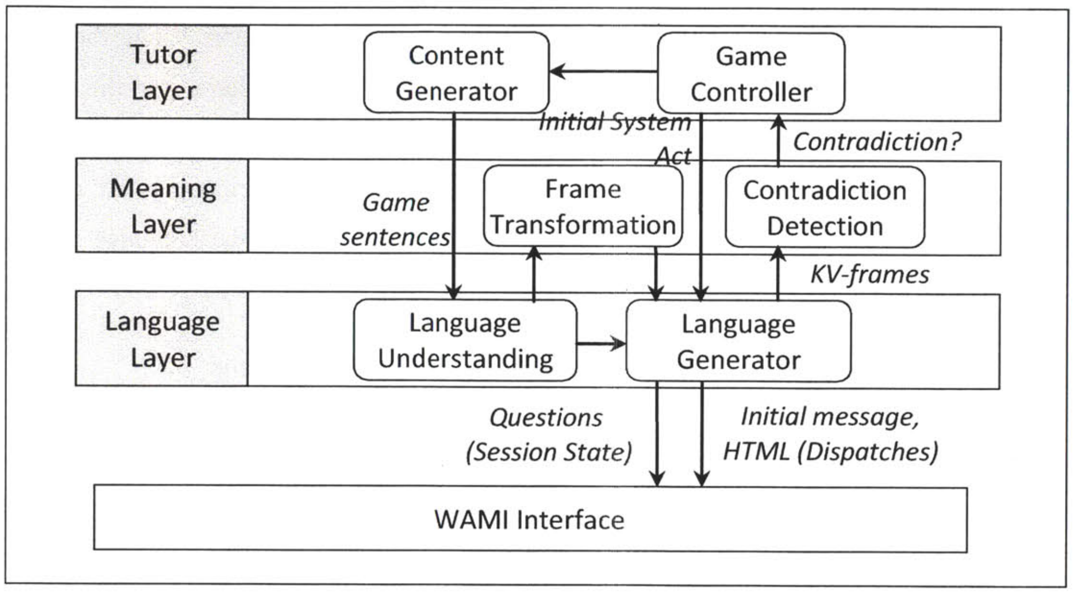


Figure 4-16. System diagram of the question-answering game (initialization).

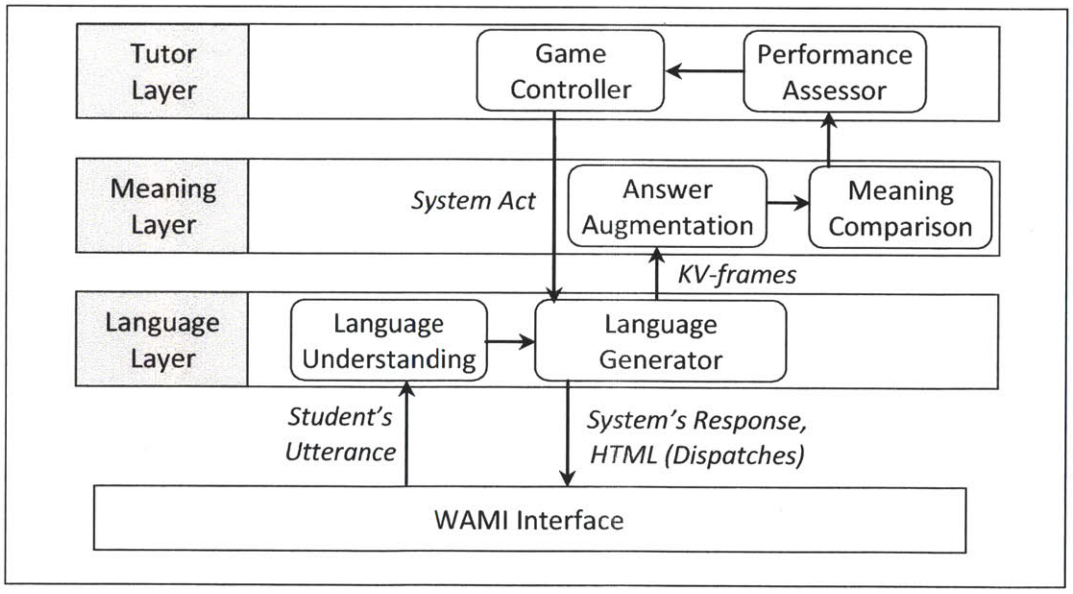


Figure 4-17. System diagram of the question-answering game (game play).

When initializing a game round, the content generator produces a list of game statements from the templates. The statements go through the language understanding and language generation modules to produce their kv-frames for contradiction detection. If any two statements contradict each other, the content generator is asked to produce another list. The process loops until a healthy list is obtained. Then, one random question is generated for each statement using frame transformation. The question list is scrambled and saved in the session state.

After all the preparation is done, the system starts to ask the first question. When the student's utterance is received, the system augments it with the question, and compares it to the statement. Depending on the comparison result and the final system act outputted by the game control module, the system either moves onto the next question, asks the student to try again, or finishes the round. The speech response and the updated GUI are generated by the language generation module, and sent to the user via WAMI. The diagram is shown in Figure 4-17.

Following is an example conversation between the system and the student.

Statement: 那只狗是黑色的 (That dog is black.)
你很喜欢吃蔬菜 (You like to eat vegetables very much)
System: 那只狗是白的吗？ (Is that dog white?)
Student: 不对，不是白的 (No, not white.)
System: 那么是什么颜色？ (So what color is it?)
Student: 是黑的 (It's black)
System: 很好。你喜欢吃蔬菜吗？ (Good job. Do you like to eat vegetables?)
Student: 喜欢 (I do. [Literally, like.])
System: 再具体一点 (Please be more specific.)
Student: 我很喜欢吃蔬菜 (I like to eat vegetable very much)

4.4 Implementation details

4.4.1 Implementing the three games as a whole

In the previous three sections, the design of the three game systems together with the critical algorithms used in the systems have been introduced. This section focuses on more implementation details.

Although the three games offer different activities and have different modules involved, they share some commonalities. The interfaces are very similar, *i.e.*, all display a list of game utterances. They are also procedurally similar. In the beginning of a round, some utterances are prepared. During the game play, the student's utterance is processed into the kv-frame representation, and is compared with the reference. The game control module uses the comparison result to output a system act. The system act is converted into a natural language response and an updated GUI via language generation. Given that they are so similar, in the actual implementation, they are implemented as a unified system using one DCTL script.

When the student accesses to the system, a login page is shown. The student can select the game mode, the prompt language, the game domain, and the level to start with on the login page. The system is implemented in the setting of Chinese learning for English speakers; thus, three prompt languages are available: English, Chinese characters, and Chinese Pinyin (the Roman representation of the characters). Three game modes are available: reading, translation and Q&A. For reading, the student can choose the prompt language to be characters or Pinyin. For translation, there is no choice for the prompt language but English. For Q&A, the student can choose one of the three choices. We developed lesson templates in multiple domains. The reading game and the translation game share the same templates in L1 English, including a set of travel domain templates and a set of flight domain templates, both of which comprise 12 lessons. The Q&A game has its own bilingual 7-lesson travel domain templates.

After the student has made the choices on the login page, the system loads up the corresponding templates and initializes the first game round. The flow chart is illustrated in Figure 4-18. Depending on the language used in the templates, a list of L1 sentences, or two lists of L1 and L2 sentences can be produced. In the situation of the monolingual templates, the L1 sentences are processed through the language understanding and language generation modules to obtain their reference translations into L2. The initialization is finished at this point for the reading game. For the other two games, the sentences in L2 are parsed and the kv-frames are stored in the session state to avoid repeated processing later during the game play. The question-answering game further uses the kv-frames to detect any contradiction, and if the sentence list is healthy, questions are generated from the parse frames.

During the game play, as illustrated in Figure 4-19, the student's utterance for the reading game is sent to a string comparison module directly. The utterance for the other two games are parsed and generated into a kv-frame, augmented for the question-answering game, and passed to a frame comparison module. The outcome of the comparison modules goes to the performance assessment and game control modules. Finally the system act is generated into a natural response and an updated HTML via the language generation module.

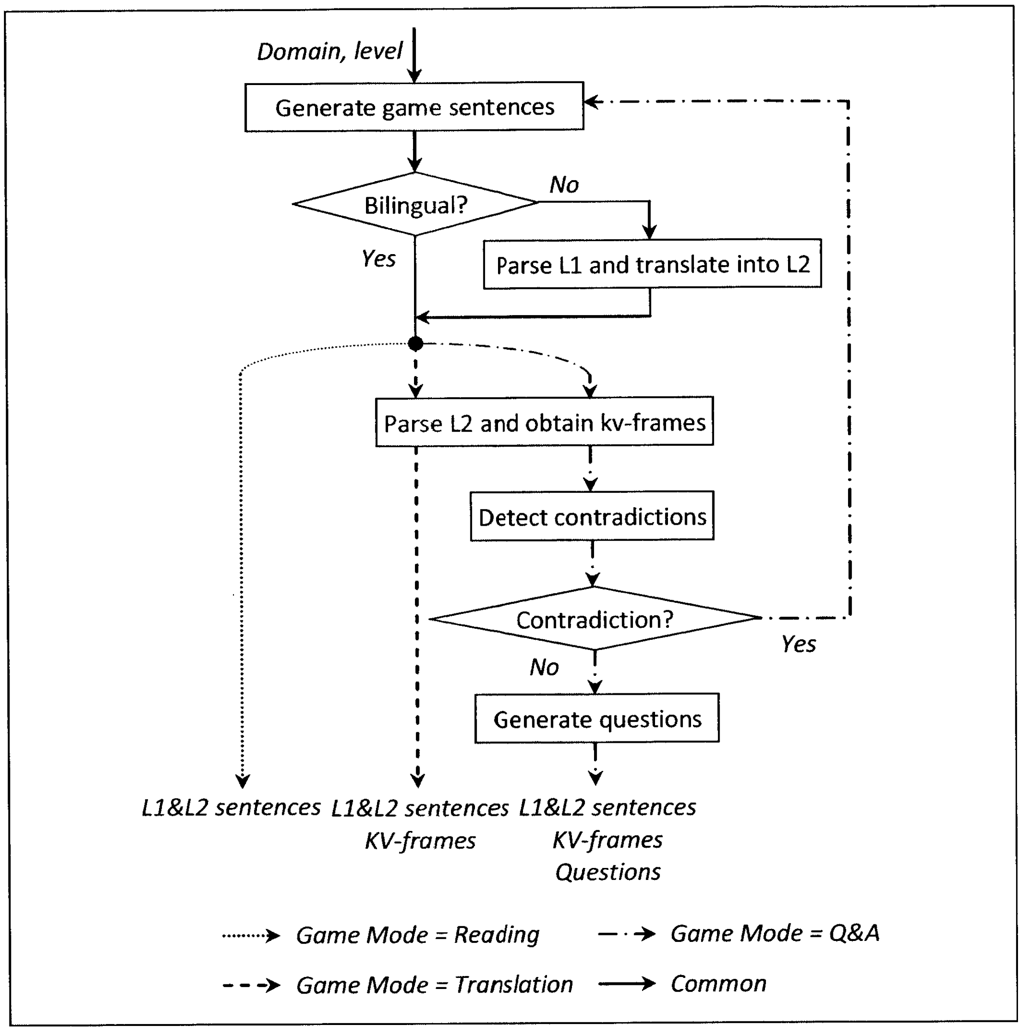


Figure 4-18. Flow chart during the initialization of a round with the three game modes consolidated into one system.

The English and Chinese grammars involved in the language understanding process are all based on generic grammars that cover a relatively wide space of sentence structures, and augmented with additional lexica according to the specified domain. For language generation, four sets of generation rules (languages) are used: Chinese, English, kv, and HTML. The language kv is used to convert a parse frame into a kv-frame, and the language HTML is used when producing an updated GUI from a system act.

In all the three game modes, the student talks in L2, *i.e.* Mandarin Chinese. A single segmental based recognizer [**60**] is used for all game modes and domains. Because the task is to recognize nonnative speech, there is a choice of the acoustic models. Obviously, it would be easier to recognize the student's accented speech by using models trained on nonnative data. Nevertheless, as mentioned previously, the systems designed in this thesis do not focus on detailed pronunciation assessment, which means for the pronunciation, the system only distinguishes intelligible (recognized correctly) and unintelligible (misrecognized). We consider using acoustic models trained on nonnative data would be a too relaxed constraint for intelligibility. Thus, the acoustic models trained on native data are adopted, so as to encourage the student to pronounce as closely as possible to the native sounds.

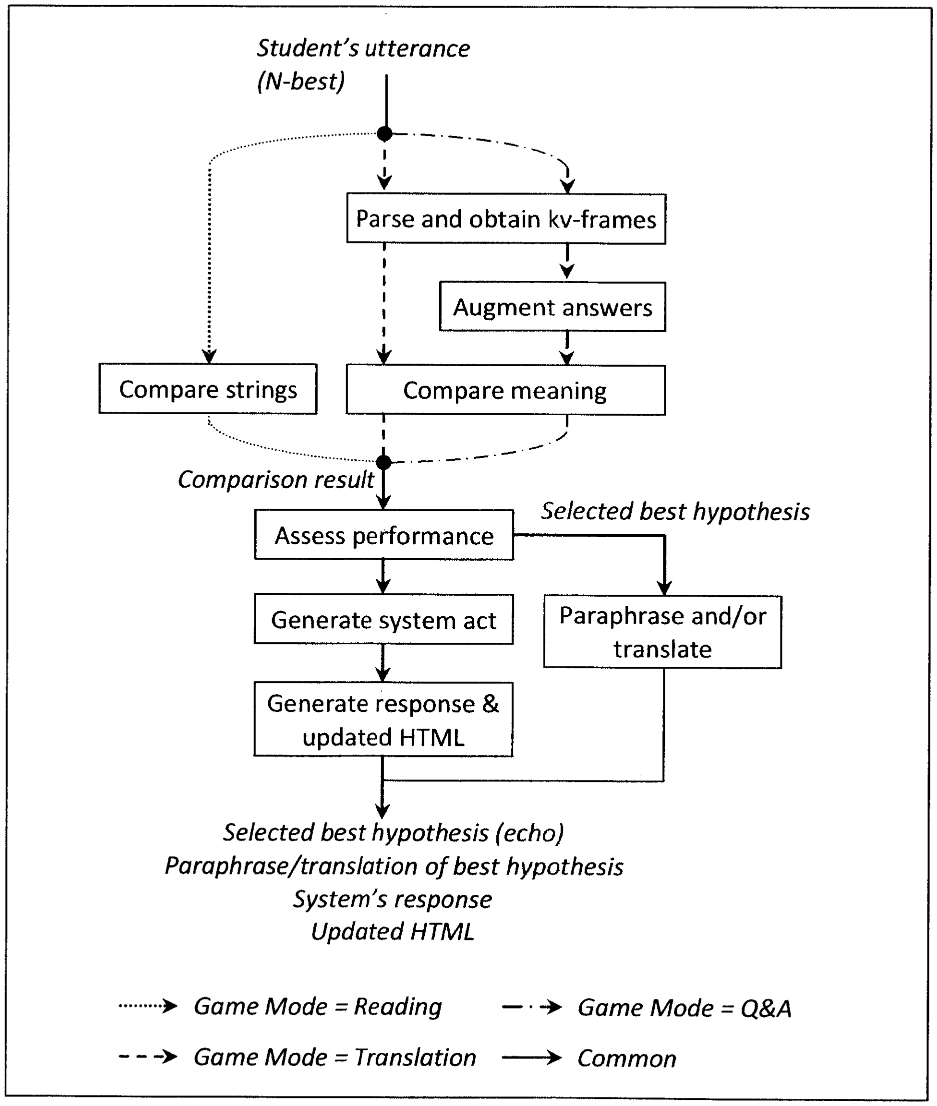


Figure 4-19. Flow chart during the game play with the three game modes consolidated into one system.

The recognizer also uses an n-gram language model to constrain the output hypothesis. The language models we use for the three games are trained on a combined corpus generated from all templates involved in the three games, as well as additional templates that model the possible utterances from the student and some generic conversational sentences. The additional templates are essentially variations of the lesson templates, for example truncating the statements to obtain short answers. The resulting size of the language model vocabulary is approximately 9.5K.

The recognizer outputs an N-best (10-best by default) hypothesis list for each utterance, ranked by a combination of the acoustic and language model score. Due to the accent in the nonnative speech, the correct hypothesis is less likely to appear as the top choice. In order to select the best hypothesis, the game context is taken into consideration. A heuristic selection method is adopted as described below. First, every hypothesis on the list is processed and compared with the game sentences one by one. If a hypothesis leads to a *match* as the comparison result, this hypothesis is selected as the best hypothesis. If no hypothesis is able to result in a *match*, the top hypothesis on the hypothesis list is selected as the best one. By doing so, the system maximizes the possibility of recognizing a correct answer from the student. Although this introduces the possibility of choosing an incorrect hypothesis that happens to match with one of the game sentences, it is imaginable that with a relatively generic recognizer of vocabulary size 9.5K, the probability of recognizing a wrong answer into a correct hypothesis is rather small.

The selected best hypothesis is echoed via a synthesizer. There are switches that enable the paraphrasing and translation of the selected best hypothesis to provide more feedback to the student on how the system interprets the input.

Two off-the-shelf synthesizers are involved in the system for English and Chinese respectively. *Dectalk* [**61**] is used to synthesize English, and a synthesizer provided by the Chinese Academy of Sciences is used to synthesize Mandarin Chinese.

4.4.2 Alternative input modality: text

Although speech is one of the major features of the games, an alternative text input modality is provided. This modality is particularly useful in two ways. When the environment is noisy and the speech recognition performance is degredated, the student can still practice other language proficiency aspects using text inputs. Besides, the text inputs also come to help when the student has trouble with one particular sound that prevents him from moving forward.

The text input box in the system accepts either L1 English or L2 Chinese. When English is typed in, it is considered as pursuing assistance. A Chinese translation is generated and shown to the student. When Chinese is received from the text input box, it is taken as the student's utterance. For this particular implementation, because the L2 language is a character-based language and is not easy to input, the text input is expected in the roman spelling form (Pinyin) with tones, and characters are automatically proposed by the system. Acknowledging that the tones tend to be very difficult for foreigners, the system provides a tone correction mechanism. The input Pinyin sequence is expanded into a word graph to include all possible tone alternatives in the lexicon. The word graph is then composed with a Pinyin-to-character finite state transducer (FST) [**57**], and a character n-gram language model FST to find the best path. The two FST used are the exact ones used in the recognizer; thus turning the Pinyin text input into characters can be viewed as a tone-less recognition process with a given phoneme sequence. The original Pinyin input is then compared with the Pinyin of the proposed characters, and any differences between the two are highlighted, in order to alert the student of the tone mistakes.

4.4.3 User database

User databases are maintained for the games. The user databases keep records of the user's scores and levels of a particular domain and game mode, and allow the user to continue playing on the level he left with last time, if he does not specify the level to start with. The user databases go with the lesson templates; *i.e.*, each set of lesson templates is associated with a database. The databases are updated at the end of each game round. The level, the number of sentences in the round, the round score, *etc.* are recorded. Although not implemented yet, it is possible to have the student access the database, and look at his performance over the time.

4.5 Evaluations of the games

We conducted the evaluation of the three games in two phases. In the first phase, we recruited several subjects to come to our lab, and gave them detailed instructions. In the second phase, we advertised our games to a list of users who are interested in Chinese learning games and asked them to play the games by accessing a public URL via the Internet. We offered them gift certificates based on the amount of data they provided. They were less instructed on the games, and they might play the game in various environments. Due to the different settings of the two phases, we provide separate analyses for the two data sets. In both phases, we focused our evaluation on the system's performance, rather than proving pedagogical effectiveness. The reading game was not evaluated, because of its simplicity and similarity to the translation game in terms of the architecture and the game procedure.

4.5.1 In-lab evaluation phase

4.5.1.1 The translation game

The two implemented domains for the translation game, *i.e.* travel and flights, were not distinguished during the evaluation. We recruited 5 subjects, 3 females and 2 males, to come to the lab. Each subject started at the first level, and was given five randomly generated utterances to translate in each round. We recorded the waveforms and the system's activity, as well as watching their behavior throughout their play. Advice was provided when they got stuck. Altogether, 615 utterances were collected from these five subjects.

We calculated the false rejection and false acceptance rate based on manual judgment. The false rejection rate was 8.6%, with almost all of the cases being caused by recognition errors. We listened to all of these waveforms and determined that most of the mis-recognized utterances were pronounced poorly or disfluently by the learners. The false acceptance rate was 0.9%. All of the false acceptances occurred when there was a minor syntactic problem in the sentence that was not identified by the system. For example, the user used an incorrect measure word for the noun. Encouragingly, we found that in the Chinese paraphrase the system gave back to the student, the syntactic problem had been automatically fixed, and we observed that the subjects did notice the implicit correction.

We calculated the average number of utterances the users spoke to complete one round, the average number of rounds they took to advance one level, and the average number of times per utterance they asked for assistance. The results are shown in Figure 4-20. The users are sorted on the horizontal axis to indicate their Chinese proficiency judged by a native speaker. The leftmost user is a native Chinese speaker. We can see that there is a good correlation between their real proficiency and the three values we measured. The users with lower proficiency tend to produce more utterances in one round, and tend to ask for assistance more frequently. The two numbers are the major factors for the system to assess the student's performance and to decide whether to adjust the game level. The result is that the poorer students tend to stay longer in the same level, as illustrated in the figure.

The game received positive feedback from the users. The users liked the feature that the system praised them, and they also appreciated the gradual introduction of new vocabulary and sentence patterns.

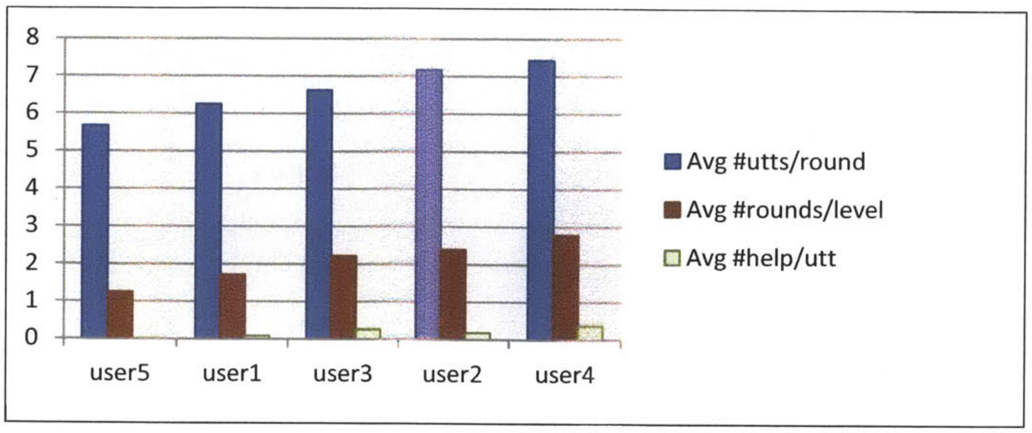


Figure 4-20. Performances of the users in the translation game. Users are arranged left-to-right in order of decreasing proficiency.

4.5.1.2 The question-answering game

The question answering game was evaluated in a similar way as the translation game, but, a simulation phase was conducted as well to evaluate the quality of the questions and the coverage of the question types. The lesson templates are composed of seven lessons. Forty frame transformation rules were written to create 17 types of questions. We simulated 42 game rounds, 6 for each lesson. In each round, 5 statements and questions were generated. We determined manually that all the questions were well-formed. The distribution of the question types is illustrated in Figure 4-21. A fair percentage of yes-no questions and wh-questions were generated in the 210 questions, and within the wh-questions, the different types of questions were distributed reasonably.

For the game system evaluation, seven subjects, 3 males and 4 females, participated in the in-lab evaluation. Three of them were native speakers. Although the participants accessed the game from different computers, we ensured that they all used a high-quality microphone in a quiet environment. A total of 732 utterances were collected from these subjects.

We categorized the utterances into three types of answers: blank-filling style short answers, such as a single yes/no or a single noun; full answers which essentially are a repetition of the statement in the list that answers the question; and other answers that are somewhere between the short answers and the full answers. The distribution of the three types, shown in Figure 4-22, is quite balanced.

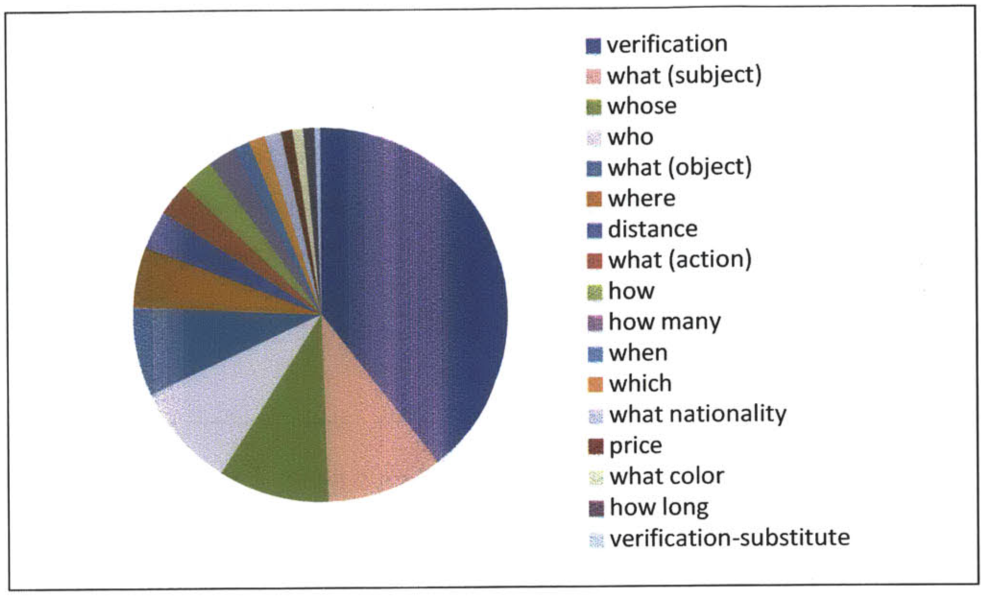


Figure 4-21. Distribution of the question types in 42 game rounds of the question-answering game.

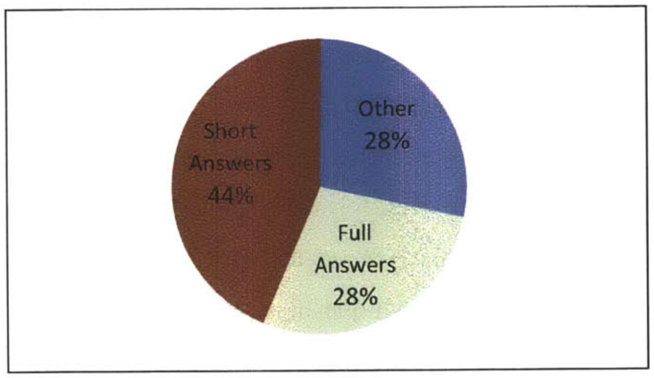


Figure 4-22. Distribution of the types of answers from 732 utterances.

In the question-answering game, the system has several different responses instead of binary choices. Due to this, we calculated the accuracy of the responses instead of the FA/FR rates. The accuracy was 91.7%, with 57 out of 61 incorrect responses caused by recognition errors. The rest of the errors were caused by ill-formed kv-frames, which were fixed before the public evaluation phase.

4.5.2 Public evaluation phase

In this phase, we opened our games to the Internet users. An email message containing the URL and some game instructions was sent to a list of possibly interested subjects worldwide. We provided awards for the subjects who completed a certain number of game rounds. The subjects were free to choose to play any of the three games they liked, as well as to select their own initial game level. The number of utterances in each round was fixed at five.

In ten days, 23 subjects accessed our games, including three subjects whose data we discarded in the analysis due to quality issues: subject #3 only provided two utterances in the middle of two game rounds of subject #2; subject #11 recorded his almost inaudible speech in an extremely noisy background; and subject #12 used a poor-quality microphone which output highly saturated waveforms and resulted in a very high recognition error that was not comparable to that of any of the other subjects. All of the remaining 20 subjects tried the translation games; 9 also played the question-answering game; and 1 also tried the reading game. The 20 subjects include 7 females and 13 males. We manually judged their Chinese proficiency on a 5-point scale based on their pronunciation and intonation. Five points indicates a native speaker, and one stands for really poor pronunciation. The average proficiency score was 3.1, with four of the subjects judged to be native speakers.

From the 20 subjects, we successfully collected 1,754 utterances for the reading/translation game, and 924 utterances for the question-answering game. We discarded 151 empty utterances and 26 utterances that the turn manager did not receive due to communication problems. We also discarded utterances related to one problematic game sentence pattern, which produced an incorrect reference translation and led to confusion. This problem was fixed after the first two days of the experiment. After pruning, we were left with 1,530 utterances for the reading/translation game, and 875 utterances for the question-answering game.

The overall sentence recognition error rate for all three games was 29.6%. Although this number is quite high, two factors played a critical role. Nearly a third (30.4%) of the mis-recognized sentences were either not a Chinese sentence, an ungrammatical Chinese sentence, or contained a totally mispronounced word. The other factor is that there were many repeated errors. When an utterance was not recognized correctly, the subject usually spoke it again, essentially repeated verbatim, and it was very likely that the second utterance would not be recognized correctly as well. To verify this theory, we calculated the rate of repeated recognition errors. We define the rate of repetition to be the total number of mis-recognized utterances divided by the unique number of mis-recognized utterances. The unique number of mis-recognized utterance with recognition errors were counted independently within each game round, so that two identical misrecognized utterances in two different game rounds are distinguished. The rate of repetition of the three games was 1.77, which means that each unique recognition error is repeated almost twice. If the repeated errors are excluded, the sentence error rate for recognition goes down to 19.2%.

The recognition error rate also varies greatly among subjects, as shown in Figure 4-23. The subjects in the plot are sorted by their human-judged proficiency. It is clear from the plot that the recognition error is influenced greatly by factors other than their nativeness, which are likely to be microphone quality and environmental noise.

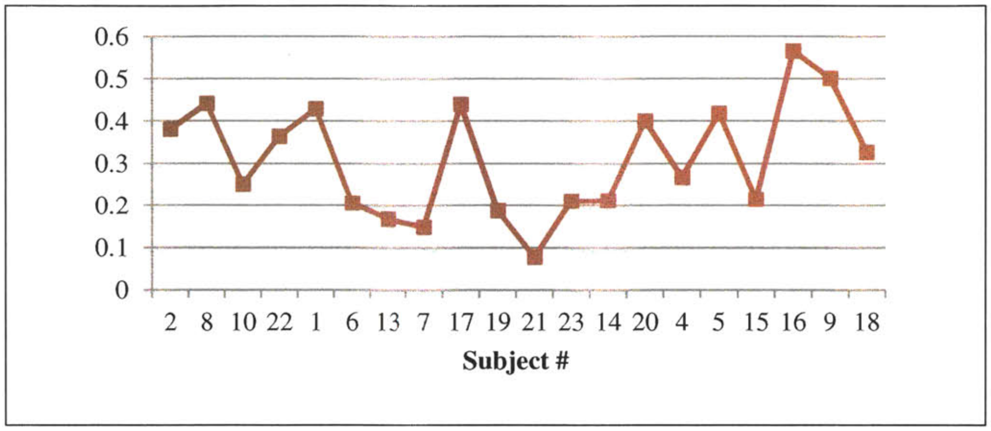


Figure 4-23. Sentence recognition error rate by subjects. Subjects are arranged left-to-right in order of decreasing proficiency.

Table 4-3. Error rates of the system responses in the public evaluation phase.

Game Genre	Error Type	Error Rate	% Caused by Recognition Error
Reading/Translation	False Acceptance	2.0%	90.3%
	False Rejection	11.6%	89.8%
Question-Answering	Incorrect Responses	9.8%	88.3%

Table 4-3 shows the error rates of the system responses. As in the in-lab evaluation, we calculated the false acceptance rate and the false rejection rate for the reading/translation game, and we did not distinguish the detailed error type for the question-answering game. We can see that the error rates were similar to those in the in-lab evaluation. Most of the errors were still caused by recognition errors. Others were mainly due to incorrect or missing information in our meaning representations. For example, "饭店" can mean either restaurant or hotel, but our kv-frame only contains one of these interpretations. Also, we did not handle verb reduplication appropriately, so that in the utterance "请帮帮我" (please help me), we treated the two occurrences of the verb "帮" as two different verbs, and falsely rejected the utterance.

In the public evaluation, it is more difficult to determine whether the subjects with poorer Chinese got more practice from simple statistics like average number of utterances they took per round. The problem is that the number of utterances per round is also dependent on environmental factors such as microphone quality and background noise level. We also notice that some subjects inexplicably repeated an already matched utterance, and thus had more utterances in each round. To take these two factors into consideration, we define a normalized average number of utterances per match as in Equations 4-7 and 4-8. In the equations, *SER* is the sentence recognition error rate, $SER_{subject}$ is the sentence recognition error rate attributed to subjects' mistakes. SER - $SER_{subject}$ gives the recognition error rate caused by other factors like background, channel, and acoustic models. Thus, a high c_{norm} means the subject recorded in a quiet environment with a high-quality microphone. On the other hand, a low c_{norm} means the subject probably used a poor recording device or played the game in a noisy environment.

$$\bar{\mu} = c_{\mathrm{norm}} \times \frac{\#\text{Total utterance}}{\#\text{Total matches}} \tag{4-7}$$

$$c_{\mathrm{norm}} = 1 - (\mathrm{SER} - \mathrm{SER}_{\mathrm{subject}}) \tag{4-8}$$

Figure 4-24 shows a plot of $\bar{u}$ for the subjects who completed at least one round of the reading/translation games. The subjects are sorted by decreasing Chinese proficiency. The logarithmic trend line illustrates that it took more effort for the lower proficiency subject to complete a match. Two anomalously low points for subject #7 and #16 result from their frequent actions of asking for assistance. They clicked the "help" button every two utterances on average, so their translations were mostly our reference translations, which were mistake-free and easy to recognize. The high value of subject #17 is due to his multiple repetition of two wrong translations which he probably thought to be correct.

For the question-answering game, we did not find a good correlation between ū and proficiency. In examining the log files, we determined that many subjects were confused with the pronoun reference of "you" and "I". Many subjects did not catch the conversational design of the game, and answered "your father is Mike" when the system asked "who is your father?". This confusion added much noise to ū, which resulted in it not being representative of the proficiency level.

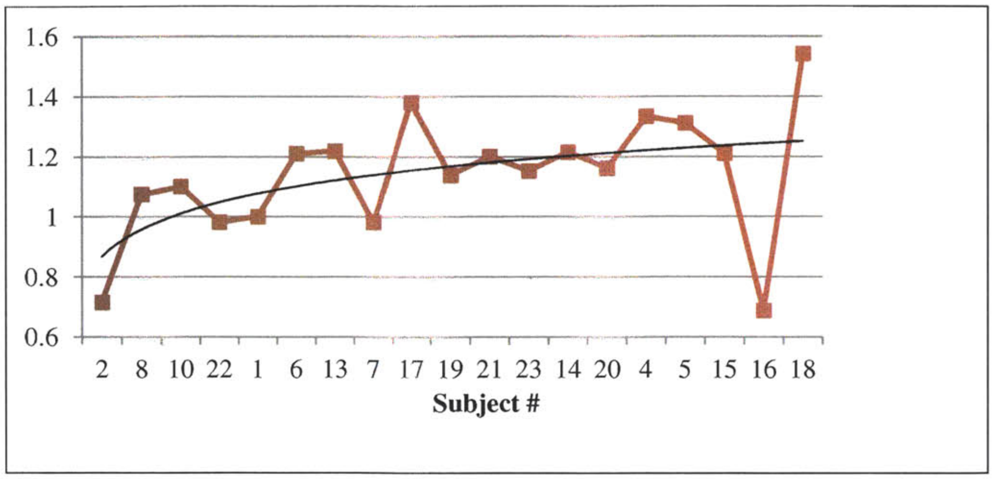


Figure 4-24. Normalized average number of utterances per match with the logarithmic trend line for the reading/translation game. Subjects are arranged left-to-right in order of decreasing proficiency.

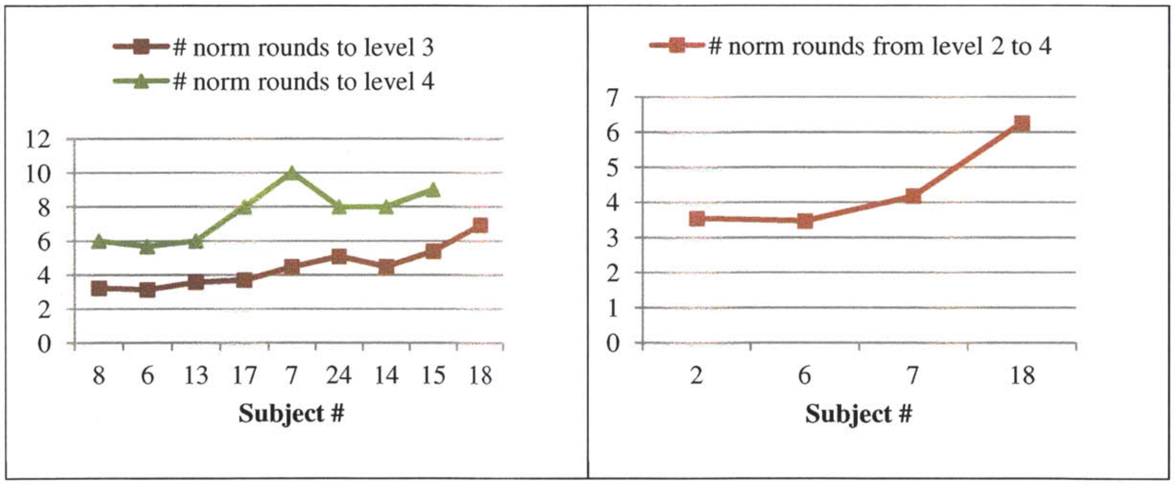


Figure 4-25. Normalized number of rounds to reach Level 3 and Level 4 for reading/translation game (left), and from Level 2 to 4 for the question-answering game (right). The subjects are sorted by decreasing human-judged proficiency.

We also analyzed how closely the system's assessment is related to the subject's Chinese proficiency. Since many subjects did not play enough rounds, and often quit the last round in a session without completing it, it is not meaningful to calculate the average number of rounds per level. Instead, we counted how many rounds they took in one game session to reach Level 3 and

Level 4 from Level 1 for the translation game. For the question-answering game, we noticed that it took the subjects one or two rounds to understand how to play the game, as well as the pronominal reference, so we discarded the information in Level 1 and counted the number of rounds they took from Level 2 to Level 4. The numbers of rounds are normalized by coefficient c_{norm} to reduce the differences in the recording conditions. The result is plotted in Figure 4-25. It can be observed from the plots that as a whole, to reach the same level, subjects with lower proficiency spent more rounds, which means that our game has a reasonable assessment algorithm. The exceptional high number for subject #7 to reach Level 4 resulted from an incomplete round at Level 3 which dropped him back to Level 2.

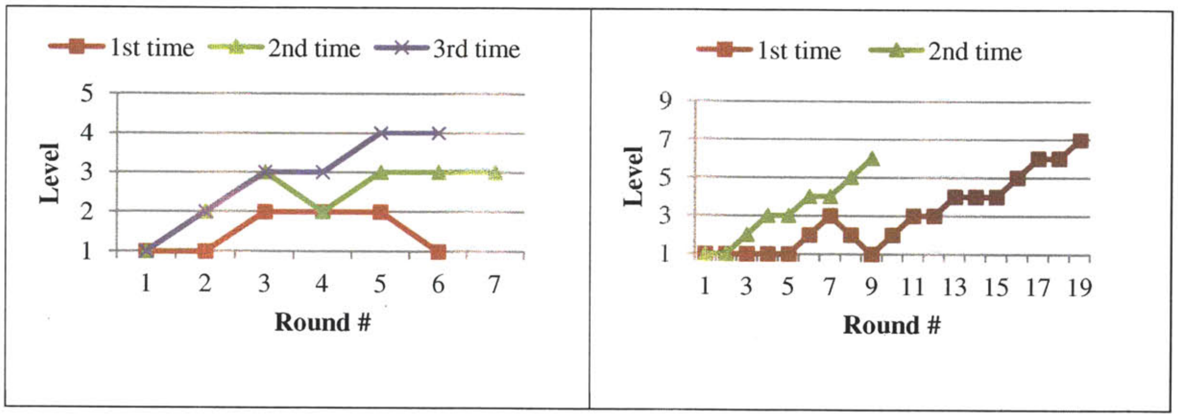


Figure 4-26. Levels subject #18 achieved in different game sessions for translation game (left) and question-answering game (right).

Several subjects accessed our system multiple times. Among them, we noticed a low-proficiency subject who played a total of 70 rounds. We found her making a lot of progress during these game plays. Figure 4-26 illustrated the levels she achieved in different game sessions. We can see that for the same number of rounds she reached a higher level when she repeated the game for a second and third time. The progress can be attributed to both increased acquaintance with the game and improvement in Chinese proficiency. For example, she had trouble with the syllable "chi" which she pronounced as "qi" causing much misrecognition. After several rounds, she realized the problem and tried hard to correct it. Finally, she learned the correct pronunciation and had it recognized correctly.

The subjects gave us considerable feedback on the games. In most of the feedback, the subjects showed their fondness for the games. Figure 4-27 shows some of the comments we received from the subjects. Most of the subjects found the games to be fun and helpful. They would like to play again and recommend them to their friends. Some of the subjects also advised that the interface should be improved to become easier for first-time subjects. Some of the subjects were very careful and pointed out mistakes in the synthesized replies. Several subjects tried to explore the space that our system is able to handle by speaking their own utterances. Their feedback was very helpful for our future development.

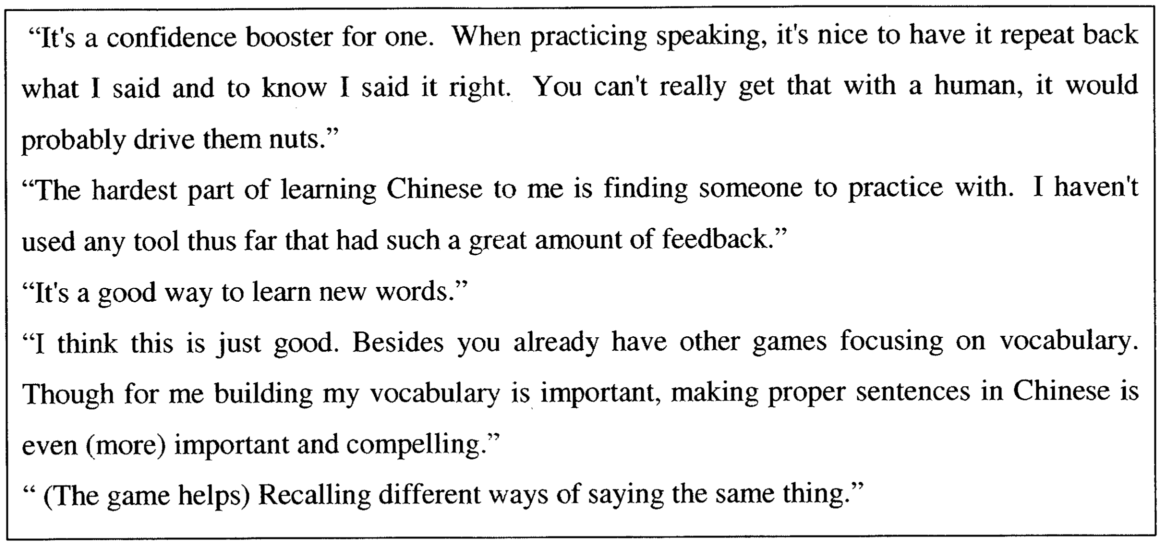

"It's a confidence booster for one. When practicing speaking, it's nice to have it repeat back what I said and to know I said it right. You can't really get that with a human, it would probably drive them nuts."
"The hardest part of learning Chinese to me is finding someone to practice with. I haven't used any tool thus far that had such a great amount of feedback."
"It's a good way to learn new words."
"I think this is just good. Besides you already have other games focusing on vocabulary. Though for me building my vocabulary is important, making proper sentences in Chinese is even (more) important and compelling."
" (The game helps) Recalling different ways of saying the same thing."

Figure 4-27. Some of the comments from the subjects.

4.6 Summary

Three games are introduced in this chapter. For a beginner level student, the reading game helps the student to be familiarized with the pronunciation and basic sentence structures of the foreign language, as well as to learn a few simple vocabulary items. The second game, the translation game, then demands the student to compose sentences in L2 by himself. In the third game, the question and answering game, the task is even more demanding. The student needs to read the L2 sentences, listen to questions in L2, and answer in L2 appropriately. All the three games use templates described in Chapter 4 to generate game contents. Without much effort, the games can share the same sets of templates to provide a smooth transition between the games.

The games are implemented as one system using the architecture we introduced in Chapter 3. We see that being implemented as a whole, though different game modes offer very different activities, many modules are shared. In other words, the modules are generic, and the games are built up by putting together these generic modules in a certain way. Indeed, these modules are general-purpose: not only designed for language learning systems, but can also be used in other applications. The extensions to the existing language understanding and language generation modules, the two-stage parsing and two-stage generation can be adopted in any parsing and generation situation. The frame transformation, which is used here to generate questions from statements, can also serve to perform other types of surgeries to any type of frames. The answer augmentation is a simplified context resolution module which can be used to resolve abbreviated answers given the question.

The three games are implemented in a setting of Chinese learning for English speakers. Templates of multiple domains are designed to show that the system and modules are domain independent. Although the system is only implemented for Chinese learning, the language independency can be observed implicitly. Since the inputs and outputs of the modules on the meaning layer are meaning representations, and the modules on the language tutor layer do not perform any language processing, we only need to examine the modules on the language layer. There are only two modules on the language layer: language understanding and language generation. In the games, the same language understanding module is used to parse the game sentences in L1, the reference translation in L2, and the student's utterance in L2. The same language generation modules are used to produce the reference translation in L2, as well as to translate the student's utterance back into L1. Therefore, we can see the modules are language-independent. Moreover, grammars and generation rules for both L1 and L2 have been developed. One can imagine that by switching the grammars and generation rules, the games can be easily inverted into English learning games for Chinese speakers.

The games were tested in both a lab setting and a public setting. In both settings, the system performed as expected most of the time. Positive feedback was received from the subjects, and we observed one particular subject who played an outstanding number of rounds and made progress during these rounds.

From the reading and translation to the question-answering, the system has increasing interaction with the student. In fact, the interaction in the question-answering game can be viewed as a one-turn dialogue. When there is a follow-up question, the dialogue develops into multi-turn. But obviously, the dialogue is very constrained and system-initiated. In the next two chapters, we will introduce a true dialogue game.

Chapter 5 Entity-Constraint-Based Dialogue Management

Dialogue may be one of the most effective ways to help a student acquire L2 language abilities; not only because participating in a dialogue involves comprehensive abilities from listening and comprehension to composition and acoustic production, but also because the dialogues are a much more real communicative use of the language. The uncertainties in a natural dialogue make the activity interesting, as the student is challenged to face all sorts of unexpected context.

Building such a dialogue system for language learning is no easy job by any means. Scripted dialogues are not the solution wanted, for they limit the student's response space to only a couple of choices. But on the other hand, allowing freer dialogues introduces difficulties on the system's side. Again, high accuracy is a key issue here. Designing a dialogue manager that is able to provide a highly appropriate response under any context is extremely difficult. Judging the student's performance would be a difficult challenge as well. But if we narrow down the type of dialogue to only goal-directed dialogues, the problem becomes more solvable in terms of both dialogue management and performance assessment.

Goal-directed dialogues refer to the dialogues in which the participants have a goal to reach. The goal is pre-defined, and is commonly understood by both dialogue parties before the dialogue. (Since we are focusing on a one-to-one learning situation, multi-party dialogues are not considered.) A typical example of goal-directed dialogues is information accessing systems, for example booking flights. In such a system, the goal is to book a flight, and both the system and the user understand the goal, so that the dialogue is carried out along a path which is by and large efficient to reach the goal. The two dialogue parties are not necessarily modeled as an agent and a customer; they may have equal status, such as two friends trying to set up an outing schedule. The goal may not be concrete, either. The important idea is that the course of the dialogue has a direction, and is predictable to some extent, which makes it different from non-goal-directed dialogues, *i.e.* chat-style dialogues.

Goal-directed dialogues are a good choice for building language learning systems. The particular goal naturally becomes the task for the student. The student is allowed to conduct natural dialogues with the system, but, because the task is given, the dialogue can be expected to move within a certain space that the dialogue manager can handle. It is also feasible to assess the student's performance in this situation, for, in addition to sentence level intelligibility and grammaticality, the distance between the current dialogue state and a successful conclusion can be measured in terms of the completeness of the goal.

Based on these discussions, a dialogue game for language learning can be outlined as follows. The student is given a task, which can be completed by conducting a dialogue with the system. The system assesses the student's performance, gives feedback and provides assistance when necessary. Here we see that the system has two distinct roles: dialogue partner and language tutor. In the reading game and the translation game, each exercise, *i.e.* reading aloud one sentence or translating one sentence, is independent and requires only one turn to complete. The system only serves as a language tutor to provide feedback and assistance. In the question-answering game, although the interaction mimics one-turn conversations, the system still behaves more like a language teacher asking the student questions, and giving feedback after the turn finishes. But in the dialogue game, the role of a dialogue partner distinguishes itself from the role of a language tutor. The dialogue partner does not need to know anything about the language learning task. It handles the dialogue as if the student is indeed seeking information, purchasing an order, *etc.* Meanwhile, in the other role, the language tutor monitors the dialogue between the student and the dialogue partner, points out the student's mistakes, and offers hints. This role does not care about how to respond to the student's utterance and carry on the dialogue. Its only responsibility is to evaluate the student's utterance in the dialogue context and provide necessary assistance.

Once the separation of the two roles is made clear, the necessary modules are also clear. For the dialogue partner role, a dialogue manager is required. For the language tutor role, a performance assessor is inevitable. Besides, a module that can produce a task, track the dialogue state, and generate possible responses on behalf of the student to serve as example utterances is also necessary. This is exactly a user simulation module. In this chapter, we will discuss the dialogue management. In the next chapter, modules that embody the language tutor role, as well as the dialogue game will be presented.

5.1 Overview

The dialogue manager is a module that takes in a user's utterance, usually in a meaning representation form, as input, and outputs a system reply, which is a system's response in a meaning representation form. More precisely, for goal-directed dialogues, the dialogue manager's functionality is to produce system replies that can help reach the pre-defined goal efficiently. As discussed above, because of the separation of the two system roles, the dialogue manager does not need to include any language learning features. It, however, needs to be generic, so as to enable development of systems in different domains. In sum, what we want to design here is a generic dialogue manager that can be used in language learning systems, as well as in other goal-directed conversational systems.

In Chapter 2, we have already mentioned the statistical approaches to dialogue management that have become prevalent in recent years, along with the problems of adopting such approaches in language learning systems. The difficulties can be captured in three aspects: heavy data dependency, controllability issues and error-recovery. The dialogue manager being discussed here not only serves the language learning systems, but it also needs to be easily usable in the language learning systems to provide highly accurate system responses with very few or even no pre-existing data. Therefore, statistical approaches are not suitable as the backbone in this case.

On the other hand, looking into the existing rule-based dialogue management approaches, systems that can handle complex mixed-initiative dialogues tend to be vey domain-dependent. The two existing dialogue systems developed under the WAMI/TurnManager architecture, the *MERCURY* system [33] and the *CityBrowser* system [41] , both can produce complex and natural dialogues, but were developed with large amounts of domain-dependent rules and code. For example, MERCURY consumes over 250 DCTL (dialogue control) rules and about 18K lines of code, which makes the maintenance extremely difficult.

Generic rule-based dialogue managers use a domain specification in addition to the generic framework to realize dialogues in a particular domain. The specification usually provides the action/task structure of the domain, as in [30] and [62]. But this approach tends to sacrifice the flexibility and forces a system-initiative configuration, as the system has a specified agenda that decides what to ask first and subsequently.

So the problem is how to make a rule-based dialogue manager more generic without sacrificing the complexity and flexibility in the dialogues. Apparently, the "abstract management and domain specification" idea is correct. But is specification alone sufficient? We consider the answer as "no". Every dialogue domain has its own unique properties and entity relations. Some may be as subtle as the definition of time periods, and others may be as complex as "the destination airport of the next flight should be initialized as the source airport of the previous flight for roundtrip itineraries, as the source airport of the first flight in multi-destination itineraries, and as blank otherwise." If all the subtleness and complexity can be expressed in one specification, the specification itself would be daunting and complicated. It would be better to supply some small amount of code to describe such details, with a mindful design to keep the domain-dependent code simple and at a minimal size.

Lastly, while the dialogue is managed in a rule-based fashion, it must be admitted that statistical methods offer great power and robustness in solving problems with uncertainty. It is useful and worthwhile to incorporate statistical inference abilities into the rule-based framework as a tool in dialogue management, as well as for possible future extensions.

The final dialogue manager, FAUNA[1], is an entity-constraint-based model supplemented with a statistical tutoring mechanism, which allows the developer to train the system using fairly natural interaction. Domain specifications are written in a declarative way, and the domain-dependent code provides customized actions for the entities in the domain. The overall framework has a unique design that treats the user as one of the knowledge sources, and multi-modality input is also taken into consideration. In the following sections, we will elucidate the different aspects of FAUNA in detail. Two implementation examples using FAUNA are given at the end of the chapter.

5.2 Dialogue-manager-centered framework

Before going into the discussion of how to handle the input and produce responses, this section examines an overall relationships among the dialogue manager, the user, and others.

When considering a dialogue management framework, it becomes apparent that the two-party dialogue involves more than the user and the dialogue manager. Information access systems are the most obvious example where databases, which store the real information, are also an essential part of the whole framework. From the user's perspective, the dialogue manager and the databases are unified as the "system". But from the dialogue manager's perspective, the user and

[1] The feminine form of *Faun* or *Faunus*, the forest goddess.

the databases are both independent identities that it needs to communicate with. The dialogue manager itself does not contain any concrete knowledge about flights, merchandise, *etc.*, for such knowledge is usually domain-dependent, and thus must be maintained outside the abstract dialogue manager. We refer to the sources that provide such knowledge as "knowledge sources" (KS). The knowledge sources are not limited to databases. They can also be servers or programs that offer computation abilities, for example to resolve an ambiguous date expression to a unique day. Given these knowledge sources, the dialogue manager's role can be better described as a mediator, which integrates the current known information, works out a plan so as to advance closer to the goal, and communicates with the user and knowledge sources accordingly.

A typical diagram of the communication between the dialogue manager with the user is shown in the upper left part of Figure 5-1. The utterance from the user in English or another natural language, is converted into some kind of meaning representation that can be understood by the dialogue manager. Upon finishing the dialogue managing, the output system reply, encoded in a similar format as the input meaning representation, is re-generated into the natural language. Thus, on the user's side, he always speaks and hears his own native language, and on the dialogue manager's side, the meaning representation is the only format for interfacing. The natural language understanding and natural language generation modules between the two serve as an interpreter to translate between the human's natural language and the dialogue manager's meaning representation.

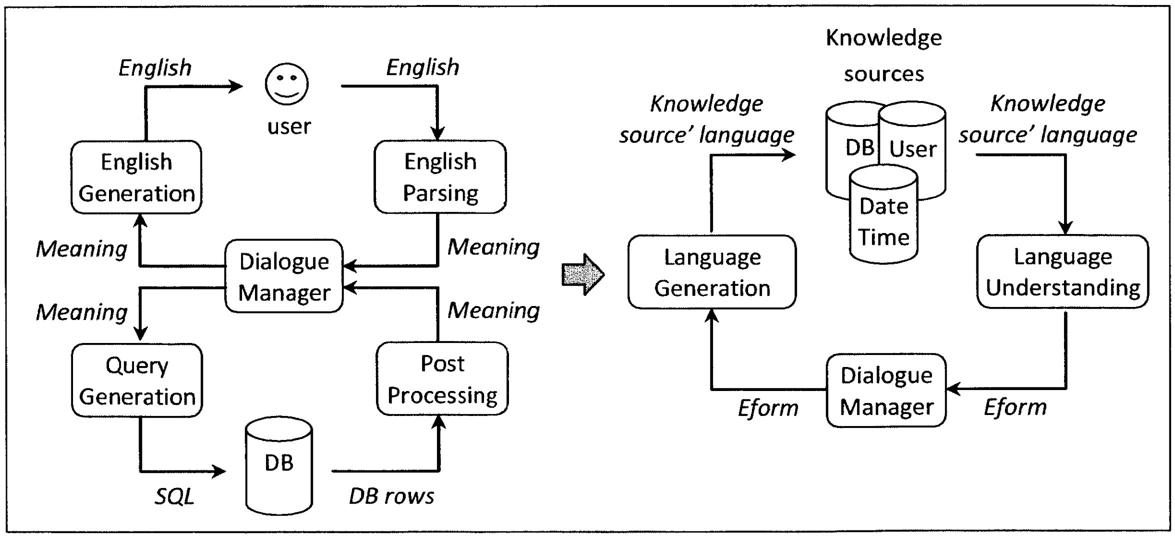


Figure 5-1. Communication between the dialogue manager, the user, and the database.

The lower left part of the figure illustrates the other communication case between the dialogue manager and the database. The dialogue manager sends a message, and a query generator converts the message into a valid database query. Before the query result reaches the dialogue manager, a post-processing routine usually exists to check the validity of the results, or perform certain reformatting.

Without much difficulty, it can be observed that the two communication cases are highly symmetric. The database can be considered to have its own language, and the query generator and the post processor work exactly the same way as the natural language generation and natural language understanding modules. Furthermore, thinking of the content in the communication, the dialogue manager receives information from the database, and it also receives knowledge from the user, though in the latter case, the knowledge is conventionally recognized as constraints. Both the database and the user have knowledge about the final goal; for example, the database stores the detailed flight information and the user knows the source and destination of the final itinerary. The dialogue manager gathers pieces of knowledge to complete a final solution. In this sense, the user can be viewed as just another knowledge source, and the communication diagrams can be collapsed into one, as shown on the right hand side of Figure 5-1.

Each knowledge source speaks its own language. The language understanding and language generation modules translate the knowledge sources' languages from/to the dialogue manager's language, "eform" (electronic form). In the implementation, each knowledge source is given a nationality, which defines the input and output language specifications. The input and output can consist of multiple perceptions. For example, the user can both hear and see the system's response, and the contents of the two can differ. The language understanding of a language can be done through parsing (TINA) or through any other specified program handlers. Similarly, the language generation can be using the standard generator GENESIS, or using other specified program handlers. Details of the implementation and the format of the specification can be found in Appendix B.

Viewing the user as one of the knowledge sources leads to a dialogue-manager-centered design. The dialogue manager receives a message from one knowledge source, compiles the information included in the knowledge source, and figures out another knowledge source to request for more information until the goal is reached. Handling the information coming from the user is ideally the same as handling the information from other knowledge sources. However, due to the implementation issues, the current version of FAUNA does need some special processing for the user, for example to signal the start and the end of a turn, and pause and wait for the user's response.

5.3 Entity-constraint-based reasoning

Having described the dialogue-manager-centered design of FAUNA, this section touches on how FAUNA internally organizes the information retrieved from the knowledge sources, and plans the dialogue according to the given domain specification.

To explain more intuitively, we will use the flight-reservation domain as the example throughout this and the following sections. The dialogue scenario in the flight-reservation domain is that the user wants to book an itinerary for a known source, destination and date. Airlines, departure time and other constraints can be specified according to the user's preference. The itinerary can be a one-way flight, two roundtrip flights or other multi-destination itineraries.

The most straightforward thinking of specifying the dialogue in a particular domain is by specifying the actions, in other words, specifying what the system should ask for and respond with in a certain condition. In the flight-reservation domain, this approach would be to say "ask for the source first. When the source is given, ask for the destination, followed by the departure date. When the three pieces of information are all collected, send a database query." It is not difficult to do so at first, but when the dialogue situation becomes more complex, this is not a preferable way, as the instructions of the actions become sophisticated very quickly. In fact, in the MERCURY system, most of the over 250 DCTL rules were written to capture the conditional actions, and after a while, they became unmanageable.

Thinking of why specifying the actions is tedious, it is soon realized that actions are consequences instead of causes. Giving explicit action specifications is equivalent to giving instructions without telling it the reason. Thus, the human developer needs to consider carefully all the possible conditions and use human intelligence to decide what to do in those conditions. But in fact, it is not impossible to tell the dialogue manager about the causes of those actions, and have it do the reasoning to decide the appropriate actions by itself.

Let us examine the flight-reservation domain again. Why does it need to ask for the source, destination and departure date of the flight? The most probable reason is that the flight database demands those fields in order to perform a reasonable search. Why does it need to ask the flight database for a search? Because that is the only knowledge source that contains information about the flights. Why does it need the information about the flights? Because the pre-defined goal in the domain is a flight itinerary, and a flight itinerary needs to have one or more flights. Going through this logic, it can be noticed that the actions are determined from two aspects: the entity structure in the domain and the constraints set by the knowledge sources. The entity structure determines that a complete itinerary must contain one or more flights, as well as that a flight contains attributes such as source, destination, and date. The knowledge source constraints determine that the detailed flight information can only be obtained from the flight database, and the source, destination and departure date must be filled before the query from other knowledge sources, *i.e.* the user. Once the entity structure and the knowledge source constraints are properly specified, the dialogue manager should be able to figure out the next action according to the current dialogue state using logical reasoning.

Based on this idea, a *declarative* specification is designed for FAUNA. All the available knowledge sources are declared, along with the knowledge they provide and the constraints called "prerequisites". The entity types are defined by elucidating the members and the complete conditions. One entity type is appointed to the goal entity. The reasoning process starts from the complete conditions of the goal entity, computes the satisfied part and not-yet-satisfied part, and issues a task according to the unsatisfied condition. The tasks are associated with action functions, which can be fully customized. In the execution of the action function, another task can be issued as a subtask or a sibling task, which creates a tree-structured task hierarchy. The leftmost leaf task is called for execution every time, until one of the stop conditions is met. Then, the system reply produced during the execution of the tasks is picked up and sent out.

5.3.1 An example

Before going into the details, an intuitive example is given below. Figure 5-2 shows an illustrative dialogue specification of the flight-reservation domain. Four knowledge sources and two entity types are declared. The knowledge sources besides the user handle the flight information, airport resolution and date/time resolution respectively. The user is assumed to know everything that cannot be obtained from any other knowledge sources, but with very high cost. The two entity types are *itinerary*, the goal entity, and *flight*. Entity *itinerary* has two member attributes: *flights* and *price*. Entity *flight* is simplified to contain five member attributes: *source*, *destination*, *date*, *airline* and *flight number*. The key :*goal* leads to the self-explanatory logical expressions for the complete conditions.

```
{c dspec
  :knowledge_sources (
        {q user
            :nation "US" }
        {q flight_db
            :nation "db"
            :knowledge ( {q flight
                              :attributes ( "flight_number" )
                              :prerequisite ( "destination" "source" "date" )
                              :handler "flight_db_query" } )
                          {q itinerary
                              :attributes ( "price" )
                              :prerequisite ( ":flights" )
                              :handler "price_query" } ) }
         {q date_time
             :nation "eform"
             :knowledge ( { q date
                              :handler "resolve_date" }
                          {q time
                              :hanlder "resolve_time" } ) }
        {q airport
            :nation "eform"
            :knowledge ( {q airport
                              :handler "resolve_airport" } ) } )
  :knowledge_source_priority ( "date_time" "airport" "flight_db" "user" )
  :nations (
        {q US
            :input_perceptions ( ":hears" )
            :output_perceptions ( ":speaks" )
            :reads {q string
                        :genesis_language "English" }
            :speaks {q string
                        :tina_grammar "English" } }
         {q db
            :input {q string
                        :genesis_language "SQL" }
            :output {q eform
                        :handler "post_process_db_result" } } )
  :goal "itinerary"
  :entities (
        {q itinerary
             :definition {c definition
                   :flights "list-of-flight"
                   :pred {p price} }
             :goal "#:flights > 0" }
{q flight
   :definition {c definition
        :pred {flight_number }
        :pred {p source}
        :pred {p destination }
        :pred {p date }
        :pred {p airline } }
   :goal "flight_number & date" } ) }
```

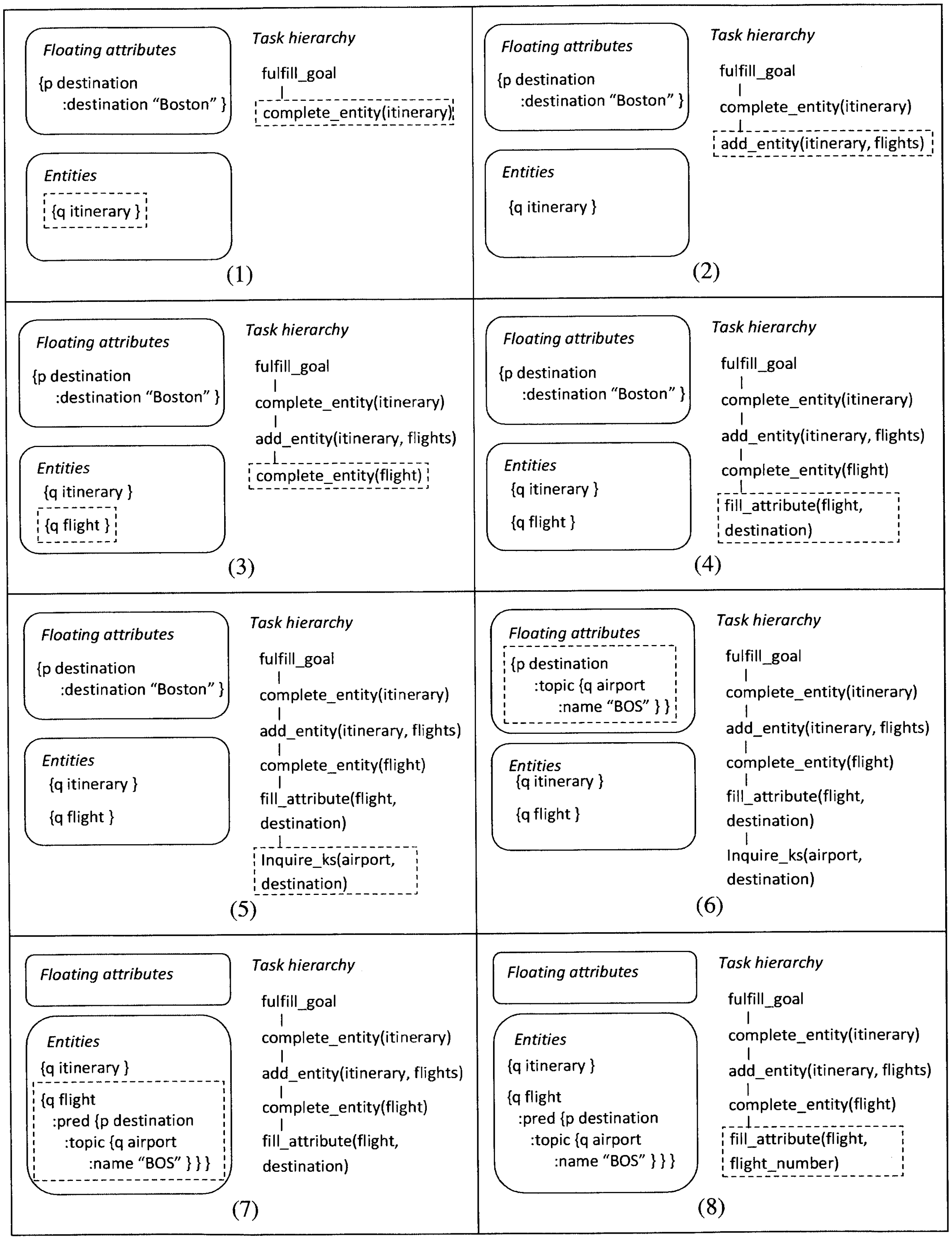
Figure 5-2. Illustrative dialogue specification of the flight-reservation domain.

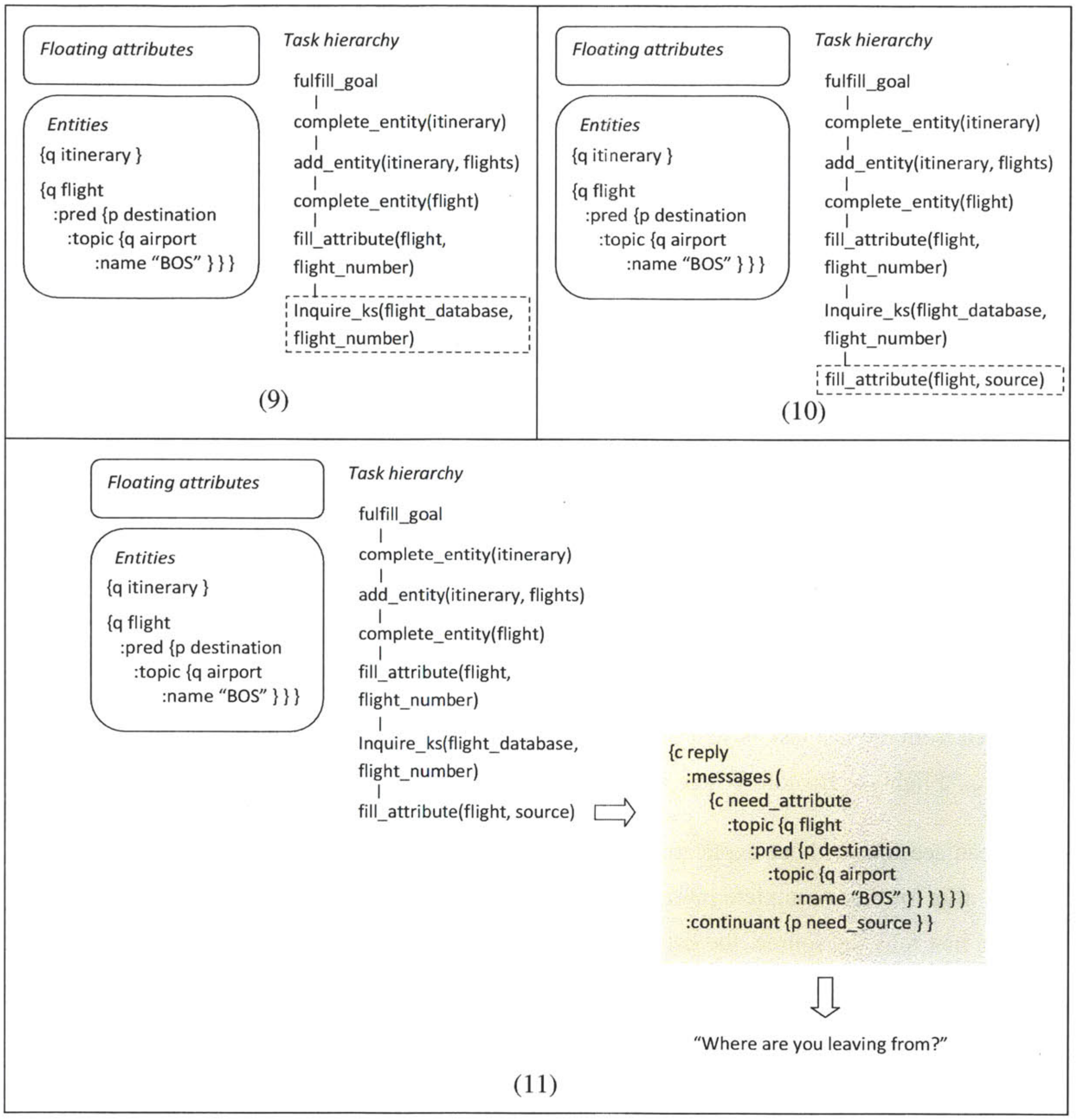
Figure 5-3 depicts the course of reasoning and planning after the user has spoken "I want to fly to Boston." Via language understanding, the English utterance is conveyed to FAUNA in language *eform* which looks like the following:

```
{c eform
   :destination "Boston" }
```

The eform input is first digested into FAUNA's internal memory state. Because *destination* is declared as an attribute of entity *flight*, a floating attribute *destination* with a candidate owner *flight*. Then the reasoning begins. A root task *fulfill_goal* is issued automatically. The following describes what happens in each step in the figure.

1. Executing *fulfill_goal*. The goal entity is *itinerary*. Since no such entity can be found in the memory state, an empty *itinerary* is created and a subtask *complete_entity(itinerary)* is issued.
2. Executing *complete_entity(itinerary)*. The completion condition "number of flights greater than zero" is not satisfied, so issue a subtask *add_entity(itinerary, flights)*.
3. Executing *add_entity(itinerary, flights)*. The type of member *flights* is a list of entity *flight*. No entity of type *flight* exists, so create a new *flight* and issue a subtask *complete_entity(flight)*.
4. Executing *complete_entity(flight)*. The floating space is first checked. It finds the floating attribute *destination*, and thus issues a subtask *fill_attribute(flight, destination)*.
5. Executing *fill_attribute(flight, destination)*. The declaration of attribute *destination* indicates a format, but the current attribute does not match the format. FAUNA finds the knowledge source *airport* which can possibly resolve this confusion, and issues a subtask *inquire_ks(airport, destination)*.


Floating attributes
{p destination :destination "Boston" }
Entities
{q itinerary }
Task hierarchy
fulfill_goal
complete_entity(itinerary)
(1)
Floating attributes
{p destination :destination "Boston" }
Entities
{q itinerary }
Task hierarchy
fulfill_goal
complete_entity(itinerary)
add_entity(itinerary, flights)
(2)
Floating attributes
{p destination :destination "Boston" }
Entities
{q itinerary }
{q flight }
Task hierarchy
fulfill_goal
complete_entity(itinerary)
add_entity(itinerary, flights)
complete_entity(flight)
(3)
Floating attributes
{p destination :destination "Boston" }
Entities
{q itinerary }
{q flight }
Task hierarchy
fulfill_goal
complete_entity(itinerary)
add_entity(itinerary, flights)
complete_entity(flight)
fill_attribute(flight, destination)
(4)
Floating attributes
{p destination :destination "Boston" }
Entities
{q itinerary }
{q flight }
Task hierarchy
fulfill_goal
complete_entity(itinerary)
add_entity(itinerary, flights)
complete_entity(flight)
fill_attribute(flight, destination)
Inquire_ks(airport, destination)
(5)
Floating attributes
{p destination :topic {q airport :name "BOS" } }
Entities
{q itinerary }
{q flight }
Task hierarchy
fulfill_goal
complete_entity(itinerary)
add_entity(itinerary, flights)
complete_entity(flight)
fill_attribute(flight, destination)
Inquire_ks(airport, destination)
(6)
Floating attributes
Entities
{q itinerary }
{q flight :pred {p destination :topic {q airport :name "BOS" } } }
Task hierarchy
fulfill_goal
complete_entity(itinerary)
add_entity(itinerary, flights)
complete_entity(flight)
fill_attribute(flight, destination)
(7)
Floating attributes
Entities
{q itinerary }
{q flight :pred {p destination :topic {q airport :name "BOS" } } }
Task hierarchy
fulfill_goal
complete_entity(itinerary)
add_entity(itinerary, flights)
complete_entity(flight)
fill_attribute(flight, flight_number)
(8)



Floating attributes
Entities
{q itinerary }
{q flight
:pred {p destination
:topic {q airport
:name "BOS" } } }
Task hierarchy
fulfill_goal
complete_entity(itinerary)
add_entity(itinerary, flights)
complete_entity(flight)
fill_attribute(flight, flight_number)
Inquire_ks(flight_database, flight_number)
(9)
fill_attribute(flight, source)
(10)
(11)
"Where are you leaving from?"

```
{c reply
   :messages (
      {c need_attribute
         :topic {q flight
            :pred {p destination
               :topic {q airport
                  :name "BOS" } } } } } )
   :continuant {p need_source } }
```

Figure 5-3. Reasoning and planning for the user input "I want to fly to Boston."

6. Executing *inquire_ks(airport, destination)*. The attribute *destination* is sent to the knowledge source *airport*, and an updated one is sent back. This task finishes and is removed from the task tree.
7. Executing *fill_attribute(flight, destination)*. This time the attribute *destination* conforms to the format. Entity *flight* adopts it, and the task finishes.

8. Executing *complete_entity(flight)*. The completion condition states "flight number and departure date are required." Neither of them has been filled. FAUNA issues a subtask according to the first unsatisfied condition, which is *fill_attribute(flight, flight_number)*.
9. Executing *fill_attribute(flight, flight_number)*. No existing *flight_number* is available from the memory space. FAUNA turns to the knowledge sources, and figures out that the *flight_database* contain this information. Thus, a subtask *inquire_ks(flight_database, flight_number)* is issued.
10. Executing *inquire_ks(flight_database, flight_number)*. The prerequisite of the *flight_database*, "source and destination and departure_date," is not satisfied; thus, a new subtask *fill_attribute(flight, source)* is issued.
11. Executing *fill_attribute(flight, source)*. When trying to find out an attribute *source*, FAUNA does not see any other knowledge source that can provide this information, and therefore has to turn to the last knowledge source, *user*. A system reply *need_attribute(flight, source)* is produced, and an *end-of-turn* is signaled.

Now that FAUNA notices the *end-of-turn* signal, it stops executing the tasks. The system reply generated from the last task is sent off to the language generator. The user then hears the English response, "Flights to Boston. Where are you leaving from?"

We can see that without explicating the actions in the domain specification, FAUNA figures out and asks for the incomplete information. Moreover, nothing more needs to be specified when the user first tells the source, the date, a combination of the constraints, or no information at all. The reasoning and planning follows similar paths.

Now we will go into more details about the declaration and dialogue execution.

5.3.2 Declaration

The declaration, or the dialogue domain specification (DSPEC), provides information about a particular dialogue domain. The DSPEC is written in the GALAXY frame format, and includes four major sections of declaration: knowledge sources, nations, entities and meta information. A complete syntax and available attributes are listed in Appendix B. Following is a brief description of each of the four sections.

Entities

Entities describe how the information in the domain is structured and related. Dialogue planning is centered around the entities, and discourse relationships are fully determined by the entity declarations. Entities usually represent objects in the real world, but can also be designed for abstract objects, analogous to the classes in object-oriented programming languages.

The declaration of an entity includes two required elements, definition and completion condition, and several optional elements: governing relationships, modifiers, customized actions and commands.

Definition. The definition lists the members of the entity. Usually, a name and a type is necessary for each member. The type can be simple types, *i.e.*, int, string, double, *etc.*, another entity type, or a list of another entity type, for example:

```
:num_flights "int"
:flights "list-of-flight"
```

A special type of members called *attribute* is declared using the predicates.

```
:pred {p destination
          :knowledge_domain "airport" }
```

Attributes are members that have complex information, but not complex enough to become an entity. The attributes can be assigned a knowledge domain, so that FAUNA can find out appropriate knowledge sources to contact when the attribute needs extra processing. If the attribute is declared as an "auto update" attribute, the corresponding update task will be issued whenever the entity has been changed.

Completion condition (goal). The goal of the entity indicates the conditions when the entity is considered as complete. It is expressed in a logical expression, for example:

```
:goal "flight_number & departure_date"
```

Modifiers. Modifiers are constraints on the attributes which have effect when there are a set of candidate entities. The only currently implemented modifier is the superlatives, for example "earliest", "latest", *etc.*

Governing relationships. The governing relationships state the discourse relationships among the members of the entity. If member A governs member B, a change in A would result in a removal of the previous B values. For example:

```
:governing_relationship {c relationship
  :flights ( "price" ) }
```

The member *flights* in the entity *itinerary* governs the member *price*, and thus if there is a change in *flights*, the existing *price* is removed. The governing relationship can be unconditioned, or conditioned on specific values of the members.

Commands. Explicit commands in the input, usually from the user, are declared to show their mapping into tasks. For example:

```
:commands {q commands
    :pred {p delete
      :tasks {q remove_entity
          :entity_name "itinerary"
          :param ":flights"
          :value "*SELF*" } } }
```

The command *delete* maps to the task *remove_entity*, which removes itself (a *flight* entity) from the member *flights* of an *itinerary* entity. One command can map to a single task or a list of tasks. The task is abstract in the declaration, and is instantiated with real entities at runtime.

Customized actions. Customized actions of the tasks for the entity are specified in a way similar to events in the modern programming languages. The actions can correspond to the reserved tasks, as well as tasks specific to the domain. If the task takes a parameter, the actions can be customized with respect to a particular value of the parameter. For example:

```
:on_fill_attribute "customized_fill_attribute"
:on_fill_date "customized_fill_date"
```

The first customization says for the task *fill_attribute*, regardless of which attribute is being filled, use the action *customized_fill_attribute*. The second customization says that when the attribute being filled is *date*, use the action *customized_fill_date*. More specific customizations have priority over more general customizations.

Knowledge sources

This section declares a list of available knowledge sources and their usage. Each knowledge source is declared with the following elements: name, type, nation it belongs to, domains of knowledge it handles, and optional initializer. The type indicates the way the knowledge source can be accessed, whether it is available in a locally linked library, or it requires external dispatches. The nation specifies the language it uses, and should be one of the nations defined in the nation section.

A knowledge source can offer multiple domains of knowledge. There are two types of knowledge: the entity-level which corresponds to the entities in the domain, and attribute-level which corresponds to the knowledge domains of the attributes. For example, the flight database provides the entity-level knowledge *flight*, and the knowledge source *airport* provides the attribute-level knowledge *airport*, which corresponds to attributes *destination* and *source* of the entity *flight*. The type of the knowledge is inferred from the name of the knowledge, *i.e.* whether the name matches an entity or the knowledge domain of certain attributes. For either type of knowledge, a function handler is necessary for FAUNA to call. It may also include prerequisites, a comparator and a summarizer. The comparator is used to compare two instances in the knowledge domain and tell their relative order. The summarizer is used to summarize a list of instances.

The user is a special knowledge source. The type is set to be "GUI". No domains of knowledge are declared under user, as the user is assumed to know everything that other knowledge sources do not know about.

The knowledge sources are ranked by priority. Higher priority implies low cost in communication. When one domain of knowledge is provided by two knowledge sources, the one with higher priority is communicated with first. Only when the first one does not provide a satisfactory result would the second one be communicated. The user is by default considered to have the lowest priority. FAUNA turns to the user only when no other knowledge sources can provide the necessary information.

Nations

Nations are the specification of the language parameters. Knowledge sources belonging to the same nation share the same set of language parameters.

Each nation has one or several input perceptions and output perceptions. The input and output are respective to the knowledge sources rather than to the dialogue manager. For each perception, format and processing parameters are declared. For example, the following nation *US*, used by the knowledge source *user*, has two input perceptions and one output perception. The two input perceptions correspond to the synthesized voice and the text display, which are generated by GENESIS using two sets of generation rules. The output perception is the user's speech, which should be parsed by TINA using grammar *English*, and further converted to a key-value representation (*eform*) using GENESIS's generation rules *dialogue*.

```
{q US
  :input_perceptions ( ":hears" ":reads" )
  :output_perceptions ":speaks"
  :reads {q string
    :genesis_language "English_text" }
  :hears {q string
    :genesis_language "English_synth" }
  :speaks {q string
    :tina_grammar "English"
    :kv_lang "dialogue" } }
```

In addition to the declared nations, an implicit nation *eform* exists for the knowledge sources that can communicate with FAUNA directly and do not require any extra language processing.

Meta information

The meta information specifies the functional keys and values in the input to FAUNA. The information includes the keys used to indicate nth, truth values (yes/no), quantifiers, as well as their corresponding values, for example the values to indicate "yes" and "no".

5.3.3 Dialogue execution

Input and reply

The input to FAUNA after language understanding and the raw reply from FAUNA before language generation are both in language *eform*, a meaning-level frame.

The content of the input eform resembles the entity structures, with the keys representing the members. The input can be a single attribute such as:

```
{c eform
   :destination "Boston" }
```

Or an explicit entity, the type of which is indicated by the key *:topic*. (Note that in an eform, the names of the top frame and subframes are all by convention "eform", and thus are not used for meaning specification.)

```
{c eform
   :topic "flight"
   :destination "Boston" }
```

Indicating an explicit entity, as in the above example, places constraints for the owners of the attributes, *i.e.*, the owner entity of the attribute *destination* is constrained to be a *flight*, whereas in the first example, the owner of the attribute is determined according to the entity declaration.

Entities can be given a nickname by using a parent key, so that in later turns, this particular entity can be referred to unambiguously using the nickname. For example, the following input expresses a *flight* entity with the nickname "return_flight".

```
{c eform
   :return_flight {c eform
    :topic "flight"
    :destination "Boston" } }
```

The input eform may also contain the key *:command* for any explicit user commands, and/or keys of meta information, such as nth, truth values, and quantifiers. Keys that do not express declared entities, attributes, modifiers, commands, or meta information are discarded.

The reply eform consists of zero, one or more messages, and a continuant. The messages represent a statement that describes the current dialogue state or the most recent action, and the continuant represents a system suggestion or request. For example:

```
:eform_to_ks {c reply_messages
   :messages (
      {c need_attribute
         :topic {q flight
            :destination {p destination
               :topic {q airport
                  :name "BOS" } } } } )
   :continuant {p need_source } }
```

Table 5-1. Pre-defined messages and continuants.

Message/Continuant	Description
No_task/None	The task hierarchy is empty.
Unable_to_proceed/None	The same task is called for execution twice in a row and thus the system enters a finite loop situation.
Show_Info/None	The message contains a certain entity /attribute
Need_attribute/need_*ATTR*	An attribute is necessary. Suggest the user to provide the attribute ATTR.
Attribute_not_understood /*ATTR*_not_understood	The attribute provided does not conform to the format declaration, and no knowledge source is able to resolve it.
No_choice/change_condition Inadequate/proceed_or_change Need_confirm/confirm_entity Need_select/select_n Need_select/too_many_options	These can be generated when a list of candidates are available for selection. If the length of the list is zero, *no_choice* is generated. If the length of the list equals the number of selection slots, *need_confirm* is generated. If list length is smaller than the number of slots, *inadequate* is generated. Otherwise *need_select* is generated. Need_select can be paired with the continuant *select_n* or *too_many_options*. The latter one appears when the candidate list is too long.
Need_confirm_action/confirm_action Need_select_action/select_action	These two are the action versions of the above. The user is asked to confirm an action (task) or to select one action from the options.

This reply contains one message and one continuant. After language generation, it can be converted into the following final output.

```
:to_ks {
   :ks "user"
   :hears "Flights to Boston. Where are you leaving from?" }
```

The messages are usually paired with the continuants. Table 5-1 Lists the pre-defined messages and their continuants. In messages containing a list of entities/attributes larger than a certain length, summarizers are called to offer a summarization of the information. In addition to the pre-defined ones, domain-dependent messages and continuants are also allowed.

Dialogue state

The dialogue state is a combination of the current entity/attribute status, the current task hierarchy, and other relevant information. These elements are stored in FAUNA's internal memory space. Each of the entities, attributes and tasks has a *unique id* number to be referred with, as well as a *turn id* to indicate its temporal position in the history.

Attributes and modifiers are extracted from the input of the knowledge sources, and remain floating until some entities pick them. The floating attributes and modifiers have certain constraints on the entities that can pick them, either derived from the entity declaration, or derived from the hierarchy of the input. When a satisfactory entity appears, corresponding tasks are issued to remove them from the floating space and put them into the owner entity. If the entity already owns an attribute with the same name, the new attribute replaces the old one, and the old attribute is put into the obsolete space.

Entities are created from the input, if there is an explicit expression of an entity, or by tasks like *fulfill_goal* and *add_entity*. When creating an entity, members with initial values are created at the same time. Later manipulations of the members of the entity are typically through explicit tasks like *add_entity*, *fill_attribute*, *etc.* The entities are stored in such a way that the current active entity is on the top, and those that have not been mentioned at the bottom, so that, when searching for an entity which meets certain constraints, the most recent entity that is mentioned is found first.

In addition to the normal entities, a list of repository entities is also maintained for entity candidates. The candidates are typically results coming back from the database that required further selection. The repository entity list serves as a cache to reduce repeated communication with knowledge sources. The query conditions are also stored, so that, if the conditions of a new query is a superset of the previous conditions, FAUNA searches in the repository entities directly rather than issuing a task to inquire the knowledge source again. The entities in this list are kept for a certain number of turns. Repository entities older than this turn limit are cleaned.

Tasks

Tasks carry out the manipulations of the entities, searches for more information and communications with the knowledge sources. The tasks are stored in a list, with each task optionally having recursive subtasks, which forms a forest structure. The execution of the tasks is a loop which executes the leftmost leaf task every time, until one of the following three conditions is met: the task hierarchy is empty, the same task is called for execution twice in a row, or an end-of-turn flag is set. Under any of these conditions, the reply eform produced during the task execution is sent out. Figure 5-4 illustrates the execution process.

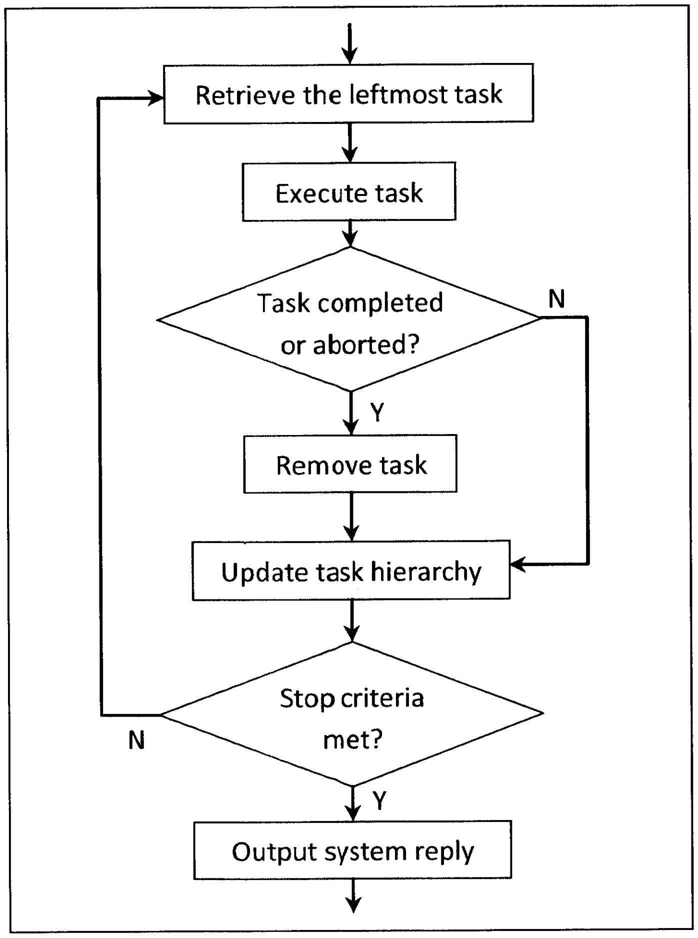


Figure 5-4. Loop of task execution.

Each task is associated with an action, either pre-defined or customized, which in turn maps to two functions in the code: a prepare function and an action function. The prepare function gathers all necessary information to perform the task, for example, to look for the concrete attribute in the task *fill_attribute*, and returns whether the task is ready to perform or not. Then the action function performs the real action, *e.g.*, changes the ownership of the attribute and updates the entity. The action function reports whether the task is completed, and if completed, how the entity has been modified. FAUNA takes advantage of the report and updates the task hierarchy by removing out-of-date tasks, and then executes the current leftmost leaf task if the stop criterion is not met.

A task can be issued as one of four types: a command task, a priority task, a subtask and a subsequent task. A command task is issued when the task is derived from a command. A priority task is issued when the task requires immediate execution. Both the command tasks and priority tasks are system-level tasks and are issued by FAUNA directly. Individual task actions are not allowed to issue these two types of tasks.

In the task actions, new tasks can be issued as either a subtask or a subsequent task. The subtask is a child task of the current task. Tasks that have subtasks cannot be completed until all of the subtasks are completed. The subsequent task is the right sibling task of the current task, which is executed after the completion of the current task.

Multiple sibling tasks can also be grouped into a task group. The task group prevents any new task from inserting between two tasks in the group. If a task in a group issues a subsequent task, the new task is added as the right sibling of the last task in the group.

During the execution of a task, a system reply can be produced. If multiple tasks generate system replies, only the latest continuant is retained. Meanwhile, all previous messages can survive as long as they do not conflict with the newer ones.

Following are some reserved tasks, most of which are pre-defined with default actions.

Fulfill_goal. This task is automatically issued at the beginning of the dialogue, and serves as the ancestor of most tasks. The task reads the goal entity of the domain, and issues a *complete_entity* task on the goal entity if the goal entity has not been completed.

Complete_entity. The task can take an entity id or an entity type. If the entity id is not provided and no entity of the specific type exists, a new entity is created. The completion condition of the entity is checked, and if the entity is not complete, appropriate tasks are issued according to the unsatisfied conditions. In the case of a conjunctive completion condition, the first unsatisfied clause translates to a new task. In the case of a disjunctive completion condition, a special *or_task* is issued, in which multiple subtasks run in parallel and the task completes if one of the subtask completes.

Add_entity, remove_entity, alter_entity. These three tasks manipulate the member entities of a parent entity. The member entities are the members of which the value is an entity type or a list of entities. The task looks for the appropriate member entity, and performs corresponding actions. In the task *add_entity*, if no such entity exists, a new entity is created, and a subtask *complete_entity* is issued on the new entity.

Fill_attribute, drop_attribute, change_attribute. Similar to the above three, these three tasks manipulate the attributes of an entity. In the task *fill_attribute*, if no suitable attribute can be found, appropriate knowledge sources are consulted, and a subtask *inquire_ks* is issued. If a new attribute already exists for the tasks *fill_attribute* and *change_attribute*, the format of the attribute is checked if applicable, and the corresponding knowledge source is consulted upon a check failure.

Update_attribute. Unlike the other reserved tasks, *update_attribute* does not have a default action. It must be customized when one or more attributes in the entity are declared as "auto update". The task is issued when the entity is modified in any way.

Add_modifier, remove_modifier. These two tasks carry out the addition or deletion of modifiers in an entity.

Inquire_ks. This task first verifies the prerequisites of the knowledge source. If the prerequisites are not met, appropriate tasks are issued as subtasks. Otherwise, the information is packed in an eform, and sent to the knowledge source together with a *session_info*. FAUNA keeps a *session_info* for each knowledge source, which can be used for the knowledge sources to record information that needs to be carried over turns.

Show_entity. This task simply produces a system reply without a continuant that contains the specified entity.

Pending objects

When there exist more than one candidate choices, or when confirmation is necessary, the system may send out the reply *need_select* or *need_confirm*. In such a case, a pending list is stored in the dialogue state. The objects in the pending list may be entities, attributes, or tasks. In the next turn, based on the input, different operations may be carried out.

a) Positive selection. For example,

- *...... Can you choose one of them?*
- *The second one.*

Or,

- *...... Do you want to add this flight?*
- *Yes.*

In this case, the appropriate item from the pending list is selected, and the pending list is removed.

b) More constraints. For example,

- *Flights from Boston to Chicago on December 12th. I have ...... Would one of them work?*
- *I want a morning flight.*

In this case, upon the incorporation of the new information into the flight entity, the pending list is also refined. After refinement, another system reply, *need_select*, *need_confirm* or *no_choice*, is generated according to the number of items left on the refined list.

c) Negative selection. For example,

- *...... Do you want to add this flight?*
- *No.*

In the case of a negative selection, the corresponding task is discarded, and the relevant entity/attribute is added into a denial list. Other possible choices are searched, and if none can be found, a *no_choice* reply is generated.

d) Other non-relevant input. For example,

- *Flights from Boston to Chicago on December 12th. I have ...... Would one of them work?*
- *I want to leave on the 15th.*

Or,

- *I have found these flights...... Would one of them work?*
- *Delete my first flight.*

In such situations, the pending list is removed, and the dialogue proceeds according to the new information or command.

Quantifiers

Quantifiers from the input eform are used to disambiguate entities or place/remove constraints on the entities and attributes. FAUNA defines five categories of quantifiers, and three of them are currently handled.

Demonstrative, *e.g.* this, that, *etc.* These quantifiers indicate an object that is being shown or has been shown in the history. For example,

I want <u>that</u> American flight.

The pending list is first looked up to see if there is any entity that matches the constraint. If not, FAUNA looks into the recently mentioned entities to find such an entity.

Other. This quantifier can act on either entity or attribute. It indicates a rejection of the current options and requests other choices. Thus, the pending list is removed, and all the items on the pending list are put into the denial list.

Any. Any can also act on either entity or attribute. For example,

```
{c eform
   :topic "flight"
   :quant "any" }
```

Or,

```
{c eform
   :departure_time "any" }
```

Any serves as a constraint remover, which drops the corresponding attribute, and/or clears the denial list of an entity.

Meta commands and exceptions

Meta commands refer to the user commands that are domain independent, such as "clear history". Since these meta commands involve operations on the history state, they are handled on a higher level using separate DCTL operations.

The meta commands are indicated in the eform using the key *:action* to be distinguished from normal commands. Currently, three meta commands are handled.

Clear history: clear the entire dialogue history and start over.

Scratch that: remove one previous turn.

Repeat that: repeat the system reply of the last turn.

Several exceptional cases are handled using separate DCTL operations, such as no parse (language understanding failure), and no reply (dialogue management failure).

5.4 Statistical inference and tutoring

As it can be seen from the last section, the dialogue execution in FAUNA is analogous to finding a solution to a problem based on the given conditions. For a goal-directed dialogue, this kind of reasoning provides an efficient guideline to achieve the goal. Nevertheless, in real life, experience also contributes in problem-solving. Such experience may be hard to express by rules, but usually can be modeled statistically. Therefore, FAUNA incorporates a statistical classification engine to support decisions which are easier to make by statistical inference.

It should be made clear that the statistical engine is designed only for specific small classification problems. A typical example is that, in some domains, certain tasks might require the user's confirmation before execution. The conditions of when confirmation is necessary can be too complicated to write down, and thus the statistical engine can help. Even though the statistical engine can be used in various places, it is not designed to solve complex problems, or to replace the overall reasoning-based framework.

5.4.1 Model and specification

The model of the statistical inference is base on the nearest neighbor model. Given a dataset X = $X^1X^2...X^n$ and the corresponding labels Y = $Y^1Y^2...Y^n$, the label of a test data point x is computed using the following equation:

$$f(x) = \underset{y_j}{\operatorname{argmax}} \sum_i \alpha^{t_i} \beta_i sim(x, X^i) \cdot \delta(Y^i, y_j) \tag{5-1}$$

where y_j indicates all possible labels, sim(•) indicates a similarity measure of the two data points, and δ(•) is the delta function which is 1 when the two inputs are identical and 0 otherwise. α and β are two coefficients in the equation. α is the temporal fading coefficient which takes value between 0 and 1, and is raised to the power of t_i, the age of the data point. Thus, older data points would have less influence. β is the correction weight, which will be explained later in this section.

The similarity measure is calculated from each feature dimension of the data points. The equation is given in (5-2), where d(•) is the distance between the two values, and w_i is the weight of each dimension, which is proportional to the number of distinct values of the feature dimension and the mutual information between the feature dimension and the labels of the data points. If the two data points are identical, the similarity is denoted by a large constant, *S*.

$$\boldsymbol{sim}(\boldsymbol{x}, \boldsymbol{x}') = \begin{cases} \dfrac{\mathbf{1}}{\sqrt{\sum_i \boldsymbol{w}_i \boldsymbol{d}^2(\boldsymbol{x}_i,\ \boldsymbol{x}'_i)}} & \boldsymbol{x} \neq \boldsymbol{x}' \\ \boldsymbol{S} & \boldsymbol{x} = \boldsymbol{x} \end{cases} \tag{5-2}$$

$$w_i \propto c(\boldsymbol{D}_i) H(\boldsymbol{D}_i, f(\boldsymbol{D}))$$

The values of the feature dimensions are not limited to numbers. Strings and lists are also acceptable. Appropriate distance functions for each feature dimension d(•) can be specified to provide meaningful measurement.

For each dialogue domain, multiple classifiers are allowed. Each classifier is given a name, and the possible labels and features are specified. FAUNA takes one or multiple frames as input to calculate the values of each feature. The features can be as simple as the value of a key in the input frame, or can be more complex and calculated by a customized function. Following is an example of the specification of a classifier called *task_needs_confirm*. The classifier takes two frames as input, and uses the name of the first frame *($core[0])*, values of two keys from the first frame *(:confirmed[0], :param[0])*, and values of two keys from the second frame *(:shown[1], :has_owner[1])* as features. The output label is either 0 or 1.

```
:task_needs_confirm {c feature
    :values (0 1)
    :ninput_frames 2
    :features ( "$core[0]" ":confirmed[0]" ":param[0]" ":shown[1]" ":has_owner[1]" ) }
```

5.4.2 Tutoring and backdoor

Developing a statistical system is usually divided into two stages. In the training stage, the models are obtained using certain amount of training data. In the test or deploying stage, the trained models are loaded to carry out the designed functionality. Data can be collected in this stage to re-iterate the training stage for a better performance. However, obtaining the initial data is not easy, especially for applications like dialogue systems. Although FAUNA's statistical engine is designed for small problems, matching data usually do not exist before the particular dialogue system is implemented.

Another problem with the training-test schema is that the system's behavior in the test stage is static. If an incorrect response is produced, the error cannot be corrected until another iteration of training is done. Furthermore, at the time of the new training process, the error might have been forgotten. It would be ideal if the statistical engine can run without initial data, be corrected immediately after any mistakes, and learn from the correction. The idea is similar to the process of teaching a child. The child is born knowing nothing, and is expected to make mistakes. But then, the adult corrects the mistake, and gradually the child learns what to do in different situations. By doing so, there is no explicit training stage, as the child can learn and perfect his behavior throughout his life, and we call this way of building statistical models "tutoring".

Using tutoring, the classifier starts with no initial data. Under that circumstance, the best it can do for classification is to guess an output. If the output happens to be correct, no measures need be taken. The features and the output label become the first data point. If, on the other hand, the guess is incorrect, a human then corrects its mistake, and this corrected data point is stored with a higher weight. For a dialogue system, when an incorrect classification result is produced, it would lead to an undesired system reply. The human tutor notices the undesired reply, rolls back one turn, and provides a correct answer. The process is illustrated in Figure 5-5.

Ideally, tutoring should be incorporated in the dialogue naturally. Nevertheless, at the current stage, we only explore the feasibility of the idea using special typed commands in a "backdoor" access to the system.

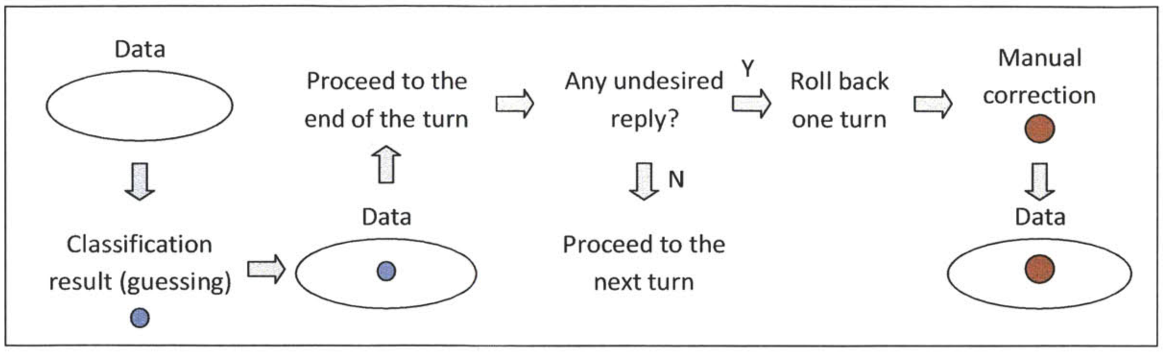


Figure 5-5. Process of tutoring.

```
This is an intervenable point.
The current situation is:
{c situation
   :conf 0.999524
   :feature_values ( 0
                     0
                     "fulfill_goal"
                     "itinerary"
                     -10000
                     0 )
   :spec {c feature
            :data_filename "../System/mercury/stat_inference_data_eval.frame.task_needs_confirm"
            :features ( ":confirmed[0]"
                        ":is_iauth_task[0]"
                        "$core[0]"
                        ":param[0]"
                        ":entity_shown[1]"
                        ":has_owner[1]" )
            :ninput_frames 2
            :values ( 0
                      1 ) }
   :unseen 0 }

My proposed result is:
0
confidence: 0.999524
bkdoor>> intervene
The current situation is:
{c situation
   :conf 0.999524
   :feature_values ( 0
                     0
                     "fulfill_goal"
                     "itinerary"
                     -10000
                     0 )
   :spec {c feature
            :data_filename "../System/mercury/stat_inference_data_eval.frame.task_needs_confirm"
            :features ( ":confirmed[0]"
                        ":is_iauth_task[0]"
                        "$core[0]"
                        ":param[0]"
                        ":entity_shown[1]"
                        ":has_owner[1]" )
            :ninput_frames 2
            :values ( 0
                      1 ) }
   :unseen 0 }

My proposed result is:
0
confidence: 0.999524
Manual solution>> 1
Input recognized as an int: 1. Is that right? (Enter to confirm, otherwise retype)
Press Enter to confirm>>
```

Figure 5-6. Correcting an incorrect statistical inference result.

The backdoor is a debugger-like tool provided by FAUNA to allow the developers to monitor the dialogue execution, as well as to provide correction for the statistical inference.

The backdoor is operated using special typed commands starting with "bkdoor". When the input is recognized as a backdoor command, it is not counted as a normal dialogue input and thus the turn id will not be incremented. When the backdoor is open, the dialogue execution may be paused or "intervened".

In the paused state, a command prompt "bkdoor>" is displayed. Developers can use commands to view the information of the current dialogue state, set or remove breakpoints, or step forward. The position where the dialogue execution can be paused is called a "landmark", which covers issuing a new task, executing a task, changing an entity, finishing a statistical inference, *etc*. New landmarks can also be added in the customized action functions. Each landmark has a label and a related object id. The developer can use a print command to see the current and past landmark list. Breakpoints can be placed on the landmarks using a substring of its label.

In the "intervenable" state, the dialogue execution can be intervened by providing manual correction for the statistical inference result. The state is enabled when the backdoor is in the tutor mode. The dialogue execution is paused after a statistical inference, and a command prompt "bkdoor>>" is displayed. The developer can view the feature values and the proposed classification result, and correct the result if it is not desired. Figure 5-6 shows a screenshot of correcting an incorrect inference result.

Table 5-2 lists some frequently used backdoor commands and their description. The commands can be typed in when the system is waiting for a normal dialogue input, or used directly when the dialogue execution is paused and the backdoor command prompt is displayed.

5.5 Supporting multi-modality

One of the recent trends in dialogue systems is that the systems are becoming multi-modal. The user interacts with the system in more than one input modality: not only speech, but also gestures. The gestures can be simple mouse clicks, or more complicated drawings. In one of the previous dialogue systems *CityBrowser* [**41**], the user is able to drawing a line or a circle on the map, and at the same time speak to the system to inquire about the restaurants near the position he draws. In order to implement such dialogue systems, FAUNA also provides multi-modality solutions.

Table 5-2. Backdoor commands.

bkdoor info	Print out verbose information during the dialogue execution.
bkdoor silent	Stop printing out verbose information.
bkdoor open	Open the backdoor.
bkdoor step	Pause at the predefined landmark actions during the dialogue execution.
bkdoor tutor [unseen] [conf *score*]	Pause at the end of statistical inference. If option "unseen" is used, only pause at unseen data points. If option "conf *score*" is used, pause only when the confidence score is below *score*.
bkdoor tutor off	Stop pausing at the end of statistical inference.
bkdoor close	Close the backdoor.
bkdoor print *type [id]*	Printing the current dialogue state, tasks, entities, *etc.* A help message is displayed if no type follows the command.
bkdoor break *name*	Set a breakpoint at a landmark action, where the label of the landmark contains *name*.
bkdoor [break] del *i*	Delete the breakpoint with id *i*.

Since the user is one of the knowledge sources, and the nationality of knowledge source defines its input and output language parameters, the multi-modality fits naturally into the handling of the nations. Each nation may have multiple output perceptions (output from the knowledge source, and thus input to FAUNA), which already promises the possibility of multi-modality. However, we shall see that this alone cannot solve the entire problem.

There are two types of multi-modality activities: independent and cross-referred. Independent multi-modal inputs stand for input that can completely express itself in a single modality. Although multiple modalities are offered in the system, for example the user can either speak or click on a map, each spoken utterance and each click forms a complete input, and the system is able to respond to the input. In this case, the speech and the mouse click can be handled easily as two independent perceptions, or even as one perception if cleverly designed.

The situation is more complex in the cross-referred inputs, when more than one modality is used to express a single meaning. For example, the user draws on a map and says "I want to go here." The drawing alone does not contain any semantic information, and the speech alone cannot be fully interpreted either. Moreover, the timing of the two input modalities is tricky. One might speak first and then click, or click before speaking. It rarely happens that the two inputs are simultaneous, since they obviously have different durations. Therefore, additional support is necessary, and parasite perceptions are defined for this purpose.

In cross-referred inputs, we first separate the main modality from other modalities. The main modality is the modality that can trigger a system response. All the other modalities are auxiliary, and are not considered as an input unless the main modality also represents an input. The auxiliary modalities are called "parasite modalities". For example, in the speech-drawing example above, the speech is considered as the main modality, and the mouse drawing is the parasite modality. Each parasite modality is given two asynchronous constraints: asynchronous lifetime and asynchronous latency. The asynchronous lifetime defines the time period an input is allowed to survive while waiting for an upcoming input in the main modality. The asynchronous latency defines the maximum latency until an input may come after the input in the main modality.

The input in the main modality may contain place holders to refer to corresponding information from other parasite modalities. Here is an example where the user first draws a line on a map and then speaks "I want to go from here to here." The perception *draws* is defined as a parasite perception, whereas the perception *speaks* is the main modality. The line is captured as two points with x and y coordinates respectively on the GUI. The input of this perception looks as follows.

```
{c eform
   :draws ( {c frame
               :point (104 225) }
             {c frame
               :point (293 300) } ) }
```

For the speech input, after language understanding, the eform looks as follows.

```
{c eform
    :speaks {c eform
        :from "<:point 1>"
        :to "<:point 2>" } }
```

The angled brackets indicate the place holders. Inside the brackets, the first token is the key that matches the input from other modalities. The second number is a temporal index, so that in the example, the value for the key *:from* should be the one that comes earlier. The indices need not be continuous, as they merely provide relative temporal order.

FAUNA processes these inputs in the following steps:

1. Check in all inputs from the parasite modalities with the current timestamp, whether they come before the main modality, after the main modality, or together with the main modality.

2. If any input remains after Step 1, they are from the main modalities. Merge them if inputs from multiple main modalities are presented.

3. If the result of Step 2 is not empty, or there is a previous incomplete input stored:

3.1 If place holders exist in the input, check out the live inputs from the parasite modalities accordingly and replace the place holders with real values.

3.2 If place holders still exist after Step 3.1, remove the keys with the place holder and flag an *incomplete_input* signal.

3.3 Proceed to dialogue execution.

3.4 If the *incomplete_input* flag is set, output the system reply with an additional note to allow for time *t* delay, where *t* is the maximum asynchronous latency of all parasite modalities. If within time *t*, a new input comes, the system reply should not be presented to the user. Otherwise, present it after time *t*.

3.5 Clear all checked-in inputs from parasite modalities.

Applying to the example above, the drawing inputs in the first eform are checked in. Since no speech input is contained in the first eform, no dialogue execution is performed. The second input is the input from the main modality, and thus the drawing inputs are checked out to fill the place holders in the speech input. The final eform that goes into the dialogue execution looks as follows.

```
{c eform
    :from  (104 225)
    :to (293 300) } }
```

5.6 Application examples

FAUNA has been used to implement two English dialogue systems in distinct domains. In both systems, the architectures are very similar. Dialogue specifications together with small amounts of code define the domains and lead to different system behaviors.

5.6.1 Flight-reservation system

Domain description

The flight-reservation domain has been used as the example in the previous sections. A more detailed description of the domain is given as follows. In the domain, the system takes the role of an agent and helps the user to find out an itinerary that satisfies the user's need. The itinerary can be one-way, roundtrip or multi-destination. The dialogue is mixed-initiative, meaning that both the user and the system can lead the dialogue. Despite the source, destination and date of departure, the user is also able to provide preferences of airlines, departure time using a specific value or an ambiguous phrase (*e.g.* earlier, morning, later in the afternoon, *etc.*) Flights can be added to or removed from the itinerary.

Implementation

Three knowledge sources besides the user are declared for the domain: one for resolving complex date and time expressions, one for looking up airport/city codes, and the third one for providing flight information. The knowledge sources date/time and city/airport are potentially useful in other domains, and thus are implemented locally as generic knowledge sources with nation *eform.* The flight knowledge source is a local simulated database with nation *db.*

The date/time knowledge source handles two types of knowledge: date and time. A specification file is loaded during the initialization to define the names of the months, names of the week days, holidays, time intervals, *etc.* The handler for knowledge *date* accepts an eform of date expression, and returns a unique date with year, month and day. If not specified, the date calculation is based on the current calendar year and the next year to come. The knowledge source is able to resolve relative dates (*e.g.* today, tomorrow, *etc.*), the holidays (*e.g.* Christmas, Thanksgiving, *etc.*), offset dates (*e.g.* four days after August second, *etc.*), and relative week days (*e.g.* next Sunday, the last Monday in May, the Tuesday after Thanksgiving, *etc.*) Once a date expression is successfully resolved, the date is record in the *session_info* as the base date. When the expression does not explicitly state the base date for offset dates and relative week days, *e.g.* four days later, the following Monday, *etc.*, the base date in the *session_info* is used for calculation. A comparator handler is implemented to provide the comparison of two dates.

The handler for the knowledge *time* accepts a time expression and resolves it into a time point with hours and minutes (*e.g.* three thirty pm), or an open time interval with one end (*e.g.* after eight pm), or a closed time interval with two ends (*e.g.* from two to four pm). Day parts such as morning, afternoon, evening are resolved into corresponding closed time intervals. For the ambiguous time expressions which do not contain explicit "am" or "pm", the knowledge source attempts to resolve the ambiguity in the case of a closed time interval by making the interval minimal and falling within the daytime. For example, the expression "from one to three" is resolved to "from one pm to three pm". A comparator and a summarizer are also available for comparing two times and summarizing a list of times respectively.

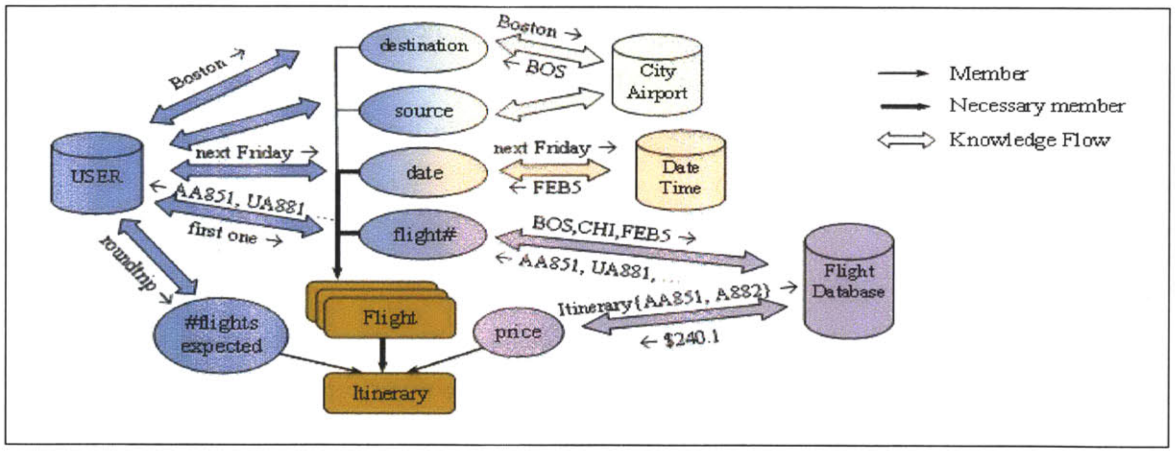


Figure 5-7. The domain logic of the flight-reservation domain.

The knowledge source city/airport accepts an eform containing the name or code of a city or an airport. The knowledge source disambiguates the input into a city code or an airport code. A comparator is defined to compare two cities or airports.

The simulated flight database generates a list of random flights at the beginning of each session. When generating the fake flights, the locations of the source and destination are carefully taken into consideration, so that the duration of the flight is close to real. The knowledge source provides the information for a single flight, as well as the price information for an entire itinerary. The handlers accept a query string, and return the appropriate flight list or the price information.

Two nations are declared. The nation *db* is defined for the flight database, which utilizes GENESIS to generate the eform into a query string for the input perception, and a post-processing handler for the output perception. The nation *US* is defined for the user. GENESIS is used for language generation (the input perception), and TINA for language understanding (the output perception).

The entity declaration includes two entity types: the goal entity *itinerary*, and *flight*. Figure 5-7 illustrates the simplified domain logic of the domain. The necessary members refer to those that compose the completion condition of the entity. The entity *itinerary* contains a list of flights, the expected number of flights which is controlled by the itinerary type, and the price, with a completion condition "#flights > 0 & #flights = #flights_expected". The entity flight contains attributes such as destination, source, date and flight number, as well as "auto update" attributes such as duration and connection time. The completion condition is "date & flight#".

The statistical engine supports the classification for the following two problems.

Task confirmation: whether a task requires the user's confirmation. For the tasks such as adding a flight or removing a flight, the system should ask for the user's explicit confirmation under some circumstances. The features used for this problem include the name of the task, the parameter of the task, whether the task has been confirmed by the user, whether the relevant entity for the task has been shown to the user, whether the relevant entity has a parent, *etc*.

Focus list: whether the user refers to the entity list or the pending list. When the user's input contains an ordinal number, it may indicate an entity on the pending list, *e.g.* "I want the second one," or it might refer to an entity on the entity list, *e.g.* "delete my second flight." The features used for this problem include the name of the command, whether the ordinal number is within the length of the two lists, whether the two lists are writable, *etc*.

Nine customized actions are defined, including four update actions for the "auto update" attributes such as flight duration and connecting time. The other five customized actions are described in Table 5-3. It can be discovered that what the customized actions handle are domain specific features, and most of them are simple extension on the default actions. The customized actions took less than 400 lines of code. Together with the implementation of the simulated flight database, all of its handler, and language processing handlers of its nation *db*, the domain specific code sums up to less than 2,000 lines. Consider an old flight reservation dialogue system [**33**] which used more than 250 rules and over 15,000 lines of domain-specific code, the new system achieved a substantially smaller size of domain-specific coding.

Table 5-3. Five customized actions in the flight-reservation domain.

complete_itinerary	Call the default action. After an itinerary is complete, issue a *fill_attribute* task to find out its price.
add_flight	Call the default action. After a flight has been added into the itinerary, set up the defaults for the next flight. For example, in the roundtrip, set the destination and source of the second flight to be the source and destination of the first flight respectively, and issue a domain-specific *ask_return_date* task to ask for the return date.
ask_return_date	Produces a system reply *need_return_date*.
fill_price	Call the default action. After the price attribute is successfully filled, produce a system reply to show the price.
fill_flight_cycle	Set the expected number of flights according to the flight cycle.

Evaluation

An evaluation is conducted in two stages: the tutoring stage and the testing stage. In the tutoring stage, a developer interacted with the system, which had no initial data, and corrected its behavior via special commands when necessary. Altogether 210 turns from 18 conversations were conducted. Figure 5-8 shows the number of manual corrections that happened during these conversations. The number of new manual corrections steadily decreased, and none was introduced during the last third of turns. The total number of manual interventions was about 7.6% of the total number of turns.

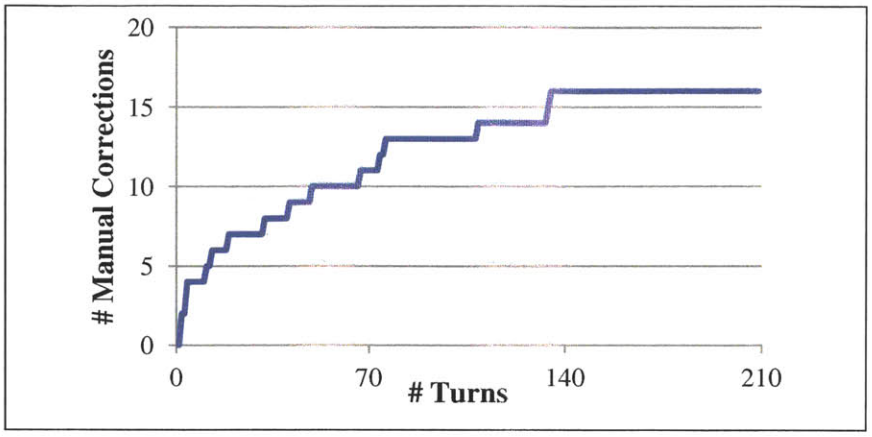


Figure 5-8. Cumulative number of manual corrections in the tutoring stage.

Table 5-4. Manual judgment of the system's replies.

Appropriate	Inappropriate due to		
	Parsing problems	Insufficient tutoring	Other causes
672 (92.2%)	20 (2.7%)	16 (2.2%)	21 (2.9%)

In the testing stage, the system was tested against a user simulator. The user simulator creates random scenarios at the beginning of each conversation, and simulates user utterances of different personality (cooperative, uncooperative, aggressive, unaggressive) using a probabilistic template based method. The details of the user simulator will be discussed in the next chapter. Fifty conversations were conducted between the simulated user and the system. The average number of turns in each conversation was 14.58, with a high standard deviation of 8.2, due to the variety of the scenario complexity and the differences in the personalities of the simulator users.

For each turn, a manual decision was made to classify the system's reply to the user into appropriate and inappropriate. The result, shown in Table 5-4, indicates that even when an uncooperative user is simulated, the system was able to handle most of the user's input appropriately. For a total of 729 turns, the system responded appropriately 92.2% of the time. One third of the failed turns were due to parse failures. Another third resulted from insufficient tutoring, where the situations were not well covered in the tutoring phase, but can be easily fixed through a few more manual corrections. The rest of the errors came from various causes. Most, however, were domain dependent, for example, the source and the destination should not be the same airport. Some other errors were due to defects in the simulator.

In Figure 5-9, an example dialogue showcases the capabilities of the system. The system is able to handle rather complex dialogues, keep track of the information (*e.g.* "the next earliest flight" after the user deletes the first flight), and manage the domain-dependent logic (*e.g.* when the user changes the source of the outbound flight, the destination of the inbound changes accordingly) with a rather small amount of domain-dependent code.

User:	i want a flight from boston to chicago
System:	Flights from Boston to Chicago. What date will you be travelling?
User:	next monday
System:	There are 5 flights to Chicago O'Hare on Feb fifteenth departing between 6 am and 5 pm. There are 4 United flights, and 1 American flight. Can you be more specific?
User:	are there morning flights from boston
System:	I have an American flight departing at 6:57 am arriving at 8:24 am on Feb fifteenth. Do you want to add this into the itinerary?
User:	how about afternoon
System:	I have a United flight from Boston to Chicago O'Hare departing at 3:50 p.m. arriving at 5:09 p.m., (other flights) … Is one of these okay?
User:	choose the earliest flight in the afternoon
System:	United flight 409 has been added to your itinerary. You depart from Boston at 12:05 p.m. and arrive in Chicago O'Hare at arriving at 1:26 p.m. What date will you be returning on?
User:	the last sunday in march
System:	I have a United flight from Chicago O'Hare to Boston departing at 9:24 p.m. on Mar 28 arriving at 12:34 am on March 29, (other flights). Please select one.
User:	delete my first flight
System:	United flight 409 from Boston to Chicago O'Hare has been removed from your itinerary. The next earliest flight from Boston to Chicago departing between noon and 6:00 p.m. on Mon Feb fifteenth I have is a United flight departing at 3:50 p.m. and arriving at 5:09 p.m. Would that work?
User:	how about from new york to chicago

System:	The earliest flight from New York City to Chicago departing between noon and 6:00 p.m. on Mon Feb fifteenth I have is a Delta flight departing from Newark Airport at 4:28 p.m. and arriving at 5:26 p.m. Would that work?
User:	okay
System:	Delta flight 481 has been added to your itinerary. (Flight summary) For the return leg, I have a Delta flight from Chicago O'Hare to Newark departing at 9:45 a.m. on Mar 28 arriving at 12:39 p.m. on Mar 28, (other two flights).Would one of these work?
User:	book the flight arriving around twelve thirty p m
System:	Delta flight 480 has been added to your itinerary. (Flight summary). You have 2 flights in your itinerary. (Itinerary summary) Do you want to find out how much your trip will cost?
User:	yes
System:	The price for the itinerary is $443.70. Is there something else you'd like to know?

Figure 5-9. An example dialogue in the flight-reservation domain.

5.6.2 Drug side-effect query system

Domain description

The second dialogue system implemented using FAUNA is a drug side-effect query system. The user asks the system about the side-effects of a drug class or a specific drug. The system looks up in the preprocessed review database [**63**], summarizes some common side-effects and presents a list of reviews which mentioned the side-effects.

The review databases currently include two drug classes, statin drugs and antidepressant drugs. The reviews are preprocessed and summarized into side-effect keywords. The side-effects are further categorized into side-effect categories, such as heart problems and cognitive problems. The user is able to query for a specific side-effect or a side-effect category, as well as frequent co-occurrence between different side-effects.

Multi-modality is involved in this system. The user has the option to inquire via speech, click on the side-effect tree to retrieve the related reviews, or click on a human body figure to access to the side-effects involving the particular body part.

Implementation

The domain logic for this system is simpler than in the flight-reservation domain. Besides the user, only one knowledge source, the review database, is involved. The knowledge source loads up an initial database at initialization, and switches to other databases when the user inquires about other drug classes. It processes eforms directly, and thus does not require any nation declaration.

The nation definition for the user is similar to that of the flight-reservation domain, including one output perception and two input perceptions. The language understanding of the output perception is handled via TINA, with a drug domain grammar augmented from the generic English grammar. The language generation of both input perceptions, *i.e.* the spoken system response and the displayed HTML, is accomplished by GENESIS. The multi-modal inputs are treated as independent inputs, *i.e.*, the mouse clicks on the side-effect tree and the body part are treated as a complete input. When there is a mouse click on the side-effect tree or the body part, the GUI sends a string containing the side-effect or the body part as an input, and the successive processing is exactly the same as if the user inputs it via speech.

User:	What side effects do Statin drugs cause?
System:	Some of the side effects that show up frequently on Statin drugs are memory problems, difficulty walking, liver damage and all body aches.
User:	Can you show me some reviews on memory problems?
System:	Some of the drugs in reviews that mention memory problems are Mevacor, Pravachol, Crestor, Lovastatin and Zocor. I've sampled a few reviews relevant to memory problems. (Top-relevant reviews are displayed on the screen.)
User:	What side effects may co-occur with memory problems?
System:	Some side effects that often co-occur with memory problems are kidney damage, muscle problems and Parkinson's disease.
User:	What antidepressant drugs can make me gain weight?
System:	Some of the antidepressant drugs that mention weight gain are Lexapro, Arnitriptryline hcl, Serzone, Anafranil and Tofrannil-pm. Other side effects in this category are weight loss, inability to lose weight and weight fluctuation. (Reviews that are most relevant to weight gain are shown on the screen as well.)

Figure 5-10. An example dialogue in the drug side-effect domain.

The entity structure is simple. One entity type *query* is declared for the domain. The entity contains attributes *drug_class*, *drug_name*, *side_effect*, *side_effect_category*, *review_posts*, *etc.*, with a completion condition that requires the existence of the *review_posts*. The customized commands are declared, *list_side_effect* and *list_cooccur_side_effects*, which correspond to two domain-specific tasks respectively. Including the actions for these two tasks, three actions in total are customized. The amount of the code for both the customized actions and the database does not exceed 2000 lines.

Evaluation

Human subjects were recruited to evaluate the system. For details of the evaluation, refer to [**64**]. Figure 5-10 shows an example dialogue in this domain.

5.7 Summary

So far, a dialogue manager FAUNA for goal-directed dialogues has been discussed. The goal of designing such a dialogue manager is to eliminate the domain-dependent effort as much as possible, and enable easy development of new dialogue systems without pre-existing data. The two applications in different domains demonstrated that it is possible to use FAUNA to build dialogue systems with a relatively small amount of domain-dependent code.

Generally speaking, FAUNA should be classified into rule-based dialogue managers. Nevertheless, the statistical engine included can be potentially used extensively, and thus transform the behavior of the system from hard decisions to a more probabilistic way. The tutoring mechanism also has much space to extend, for example to be utilized to personalize the system behavior to different users.

FAUNA is developed for general purposes. Undoubtedly, dialogue systems for language learning are among the applications. More precisely, as the roles of the system in the language learning dialogue game have been divided into a dialogue partner and a language tutor, FAUNA can be used in a straightforward way as in other conventional dialogue systems.

In the next chapter, we turn to the other role of the system, the language tutor. Two modules, the user simulator and the performance assessor are discussed.

Chapter 6 The Fourth Game: Dialogue

In a language learning system, being able to converse alone is not enough. The difference between a conventional dialogue system and a dialogue system for language learning is analogous to that between an average foreigner and a foreign language teacher. It is highly likely that the student and the system end up with mutual misunderstanding if it is merely a conventional dialogue system designed for native speakers. Therefore, the other role of the system, the role as a language tutor is equally important.

In the beginning of the previous chapter, the functionalities of the system as a language tutor were mentioned. Briefly speaking, this role needs to provide appropriate assistance when the student encounters difficulty during the dialogue, and to evaluate the student's performance from a language learning perspective. Providing assistance in a dialogue activity is different from what was required in the previous three games. As the most likely situation of a student asking for help in a dialogue is that he does not know how to proceed in the current dialogue state, the assistance must be dynamic according to the current dialogue state. The system should provide example sentences from the standpoint of the student, which is exactly a user simulation process. We will introduce a user simulator in Section 6.1.

Assessing the student's performance in a dialogue is also an important problem. A simple pass/fail decision at the end of a dialogue is not adequate and convincing. We will discuss the heuristic assessment module in Section 6.2. The discussion of the user simulator and performance assessor will be followed by a description of the implementation of the game system in Section 6.3, as well as experiments and evaluations conducted with the game system in Sections 6.4 and 6.5.

6.1 User simulation with personalities

Techniques in user simulation have been investigated for many years in the conventional dialogue system area. The two major uses of a user simulator are to test a dialogue system and to generate training data for a data-driven dialogue system. Compared to dialogue management, the behavior of the user typically contains more randomness and variance. To model such randomness and cover a large variety, statistical approaches have been prevalently used in user simulation, such as *n*-gram based simulation [**65**] [**66**], graph based simulation [**67**] [**68**], Bayesian network based simulation [**69**], and HMM-based simulation [**70**]. In most research work, the user goal is assumed unchanged throughout the conversation. Simulation performed on the intention-level is able to capture the consistency of the user intention. However, the style, or the personality of the user is not guaranteed to be consistent. For instance, the simulated user may appear very cooperative at first, and suddenly switch to behave very uncooperatively later.

In a dialogue system for language learning, a user simulator has another kind of use. As the simulator maintains a consistent user intention and simulates the user behavior based on the intention, it can naturally be used in two ways. First, the method that produces a user intention for a simulation is similar to the one needed to generate a dialogue task for a student to solve. Second, the behavior simulation can serve as a helper when the student asks for assistance. In order to provide meaningful assistance, *i.e.*, the example sentences for the students, we would like to control the style of the simulated user. Generally in a language learning system, we would like the simulated user to behave normally and cooperatively. But there might also be cases where simulating ill-behaved users are useful. For example, the responses from ill-behaved simulated users could be mixed with the appropriate ones, and the student could be asked to distinguish between them. At the same time, simulating ill-behaved users can also serve as a stress test that is very useful for developing dialogue systems.

To produce such different types of simulated user, we design a user simulator that explicitly models the personality of the user. The simulator generates a user intention, or a scenario, based on provided templates and post processing instructions. During the dialogue, the simulator accepts the dialogue manager's reply, updates its internal memory state, and generates a response from the response templates based on the personality and the current conditions.

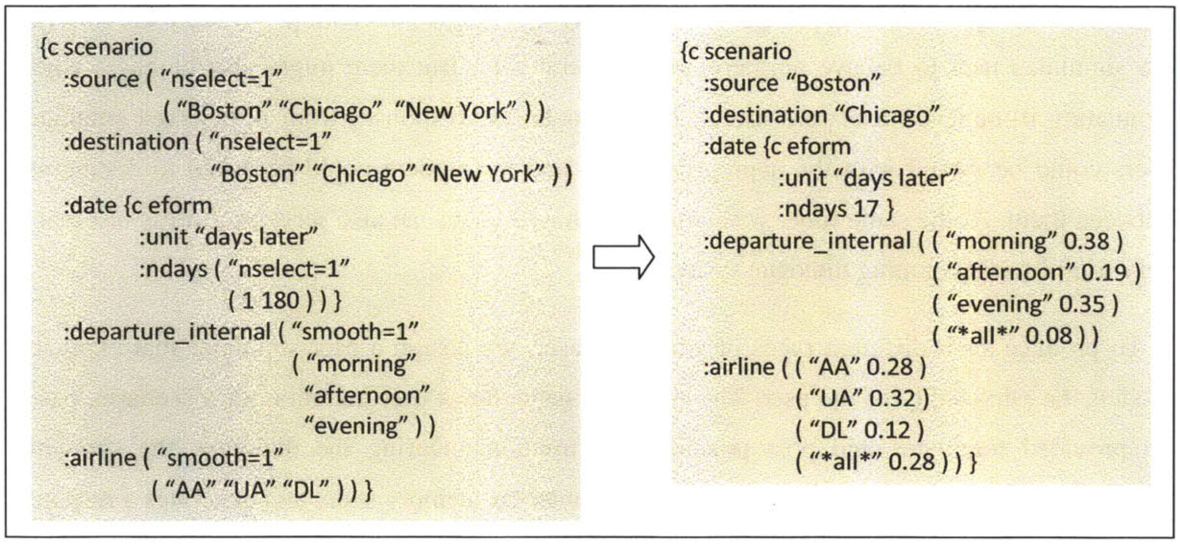
The user simulator is specifically designed to work with the dialogue manager FAUNA, described in Chapter 5. The input and output formats are designed according to FAUNA's specification. It takes the eform representation of FAUNA's reply, and produces a user's response also in eform format. It also shares some of the knowledge source handlers with the corresponding dialogue system to allow complex user intention generation.

In this section, we will describe the simulator in a generic way, focusing on the scenario (user intention) generation and personality-incorporated response generation. The special use in the language learning system will be described later in the implementation of the game system in Section 6.3.

6.1.1 Scenario generation

The scenario represents the user intention and preferences. For example, in the flight reservation domain, a scenario expresses when to travel and where to go. The user simulator needs to maintain a scenario so as to produce consistent user responses, but on the other hand, the scenario can still be changed during the conversation as if the user changes their mind.

To produce a scenario, the developer provides a list of scenario templates. The user simulator loads the templates and randomly picks one to instantiate. After that, a check is performed to ensure the instantiated scenario satisfies certain criteria. Additional post processing is also allowed if specified.



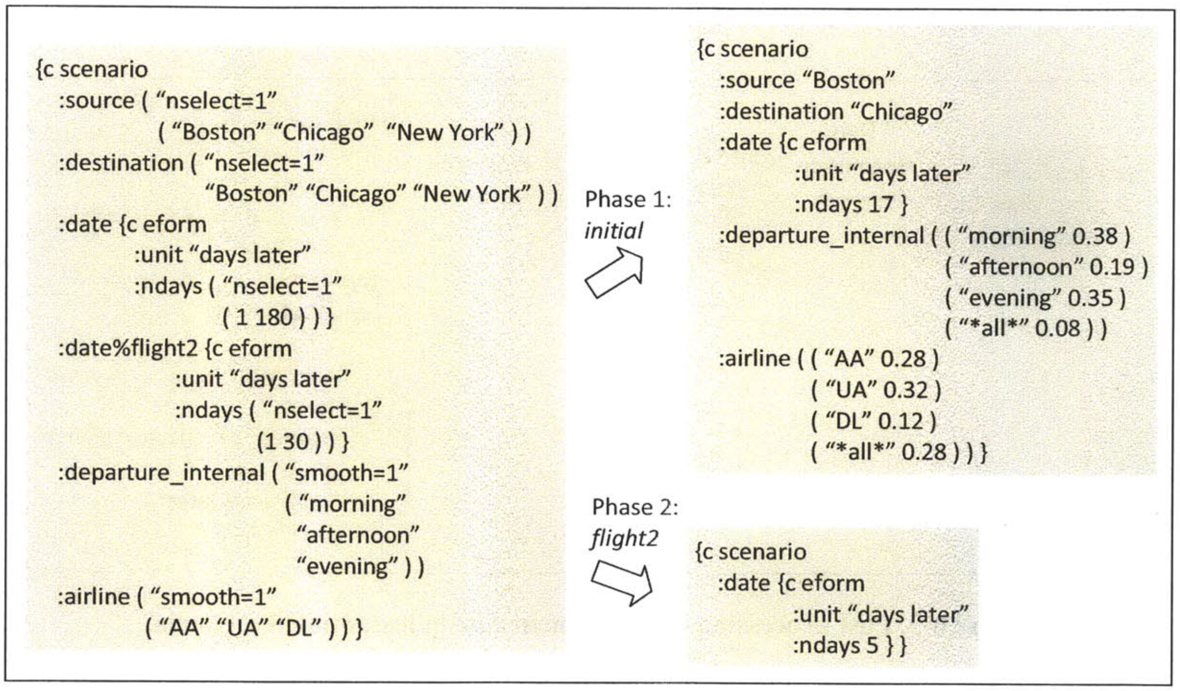
```
{c scenario
  :source ( "nselect=1"
            ( "Boston" "Chicago" "New York" ) )
  :destination ( "nselect=1"
                 "Boston" "Chicago" "New York" ) )
  :date {c eform
         :unit "days later"
         :ndays ( "nselect=1"
                  ( 1 180 ) ) }
  :departure_internal ( "smooth=1"
                        ( "morning"
                          "afternoon"
                          "evening" ) )
  :airline ( "smooth=1"
             ( "AA" "UA" "DL" ) ) }
```

```
{c scenario
  :source "Boston"
  :destination "Chicago"
  :date {c eform
         :unit "days later"
         :ndays 17 }
  :departure_internal ( ( "morning" 0.38 )
                        ( "afternoon" 0.19 )
                        ( "evening" 0.35 )
                        ( "*all*" 0.08 ) )
  :airline ( ( "AA" 0.28 )
             ( "UA" 0.32 )
             ( "DL" 0.12 )
             ( "*all*" 0.28 ) ) }
```

Figure 6-1. An example of scenario generation.

Figure 6-1 demonstrates a simple scenario template together with a possible instantiation in the flight-reservation domain. The template consists of a set of elements, an enumeration of all the possible values of the elements, and the instantiation parameters for the elements. During generation, each element is instantiated into a single value or a list of probabilistic values according to the parameter. In the example, the elements *source*, *destination* and *date* are instantiated into a single value following the parameter "nselect=1". The element *departure_interval*, however, is instantiated into a list with a smoothed probability distribution over the values, which represents the user's preferences over these choices. The special value "*all*" stands for no constraint on this element. The following steps describes the instantiation procedure.

1. *For each possible value, assign a random probability.*
2. *Normalize the probabilities to sum to one.*
3. *Smooth the probability distribution by raising each probability to the n^{th} power, where n is the smoothing parameter. Normalize the probabilities.*
4. *If parameter "nselect" is specified, or the probability of one value exceeds a single-value threshold, choose the top "nselect" values. Otherwise, add the special value "*all*" and renormalize the probabilities.*

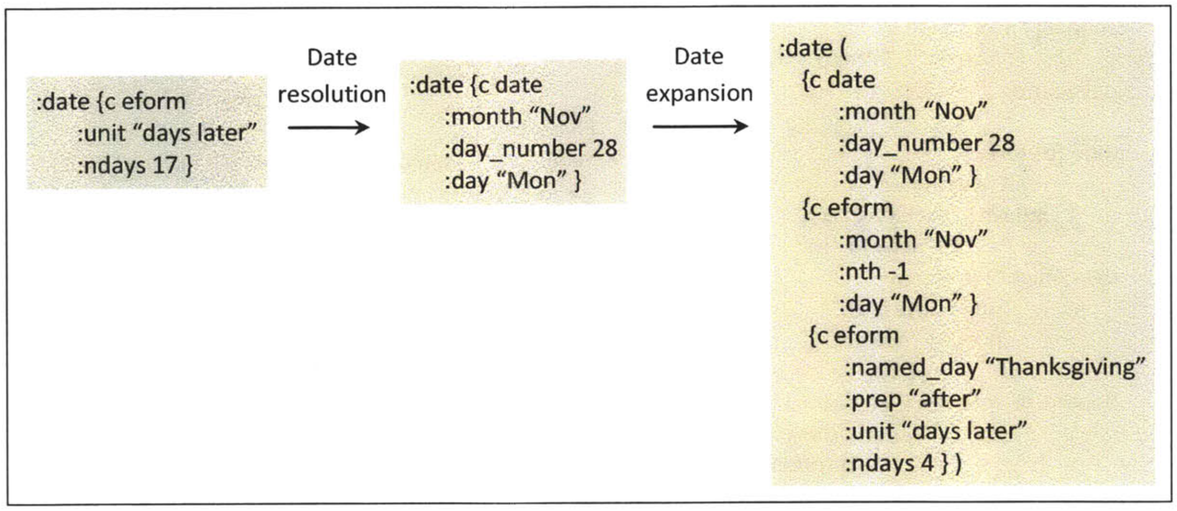

```
{c scenario
   :source ( "nselect=1"
            ( "Boston" "Chicago" "New York" ) )
   :destination ( "nselect=1"
            "Boston" "Chicago" "New York" ) )
   :date {c eform
          :unit "days later"
          :ndays ( "nselect=1"
                  ( 1 180 ) ) }
   :date%flight2 {c eform
          :unit "days later"
          :ndays ( "nselect=1"
                  (1 30 ) ) }
   :departure_internal ( "smooth=1"
                  ( "morning"
                    "afternoon"
                    "evening" ) )
   :airline ( "smooth=1"
          ( "AA" "UA" "DL" ) ) }
```

```
{c scenario
   :source "Boston"
   :destination "Chicago"
   :date {c eform
          :unit "days later"
          :ndays 17 }
   :departure_internal ( ( "morning" 0.38 )
                         ( "afternoon" 0.19 )
                         ( "evening" 0.35 )
                         ( "*all*" 0.08 ) )
   :airline ( ( "AA" 0.28 )
              ( "UA" 0.32 )
              ( "DL" 0.12 )
              ( "*all*" 0.28 ) ) }
```
```
{c scenario
   :date {c eform
          :unit "days later"
          :ndays 5 } }
```

Figure 6-2. An example scenario with phases.

In a more complex situation, the scenario can be written to contain phases. Phases are useful when an element carries different values in different stages of the dialogue. For example, in an itinerary with multiple flights, the elements *source* and *destination* have different values when the conversation enters different stages. An initial phase **init** is presumed. Each phase has an entering condition and a finish condition. Each element can be specified to belong to a particular phase using a "%" mark followed by the phase name, or be phaseless by listing it in an *out_phase* list. An element which neither has a phase indication nor is on the *out_phase* list is considered to belong to the initial phase. An example is given in Figure 6-2, which describes a roundtrip itinerary. In the phase flight2, the scenario only contains one element *date*, as no information other than the return date needs to be conveyed to the dialogue system in the case of booking a return flight.

After the scenario is instantiated, a check against the scenario constraints, specified in a logical expression, is performed to ensure the scenario is healthy, *e.g.*, the destination is not the same as the source. If the instantiated scenario fails to satisfy the constraint, a new scenario is instantiated until the constraints are satisfied. Post processing handlers, which are the knowledge source handlers defined in FAUNA, are called to carry out further modifications on the elements of an instantiated scenario. Figure 6-3 shows an example, where the *date* goes through a date resolution followed by an "expansion" to produce a list of alternative expressions for that particular date.



```
:date {c eform
    :unit "days later"
    :ndays 17 }
```

```
:date {c date
    :month "Nov"
    :day_number 28
    :day "Mon" }
```

```
:date (
  {c date
      :month "Nov"
      :day_number 28
      :day "Mon" }
  {c eform
      :month "Nov"
      :nth -1
      :day "Mon" }
   {c eform
      :named_day "Thanksgiving"
      :prep "after"
      :unit "days later"
      :ndays 4 } )
```

Figure 6-3. Post processing of the element *date* in instantiated scenario.

6.1.2 Memory state

After a scenario is generated, it is stored in the simulator's memory state. The memory state stores the user's knowledge, as well as information obtained from the dialogue manager, such as the flight that has been booked.

When a dialogue manager's reply is received by the user simulator, the memory state can be modified using some conditioned rules. Following is an example that states that, when the reply is *flight_added*, the value of the key *:topic*, which is the newly added flight, should be added into the key *:*itinerary** in the memory state.

```
{c rule
   :conditions ":*reply* flight_added"
```

```
        :save ":topic"
        :into ":*itinerary*" }
```

6.1.3 Template-based response generation

Given a generated scenario, upon the receipt of the dialogue manager's reply, the user simulator needs to produce a user response. The choice of the response depends on multiple factors: the dialogue manager's reply, the user's current knowledge, *i.e.*, the memory state, and the user's personality. We use the word "strategy" to describe how these factors would have impact on the user response.

The strategy is expressed by a list of rules. The rules describe the most preferred way to respond in a certain situation, as well as the most likely way for a user with a particular personality to respond. More specifically, each rule consists of a condition and a list of response templates. Each response template is instantiated into a response according to the scenario, and assigned a score based on the personality description in the template and the condition test against the dialogue system's reply and the current memory state. After all the response templates are instantiated, one is chosen by random sampling based on the scores.

In this subsection, we will first introduce the response generation without personality, including the condition test, the response contents and actions. The next subsection will focus on incorporating personality features into the response generation.

Condition test

Each strategy rule optionally specifies a condition. The condition is test against the combination of the dialogue manager's reply and the current memory state. The testing produces one of the following four results.

Result 1: not satisfied. The condition is not satisfied.

Result 2: no condition. The rule does not contain a condition.

Result 3: weak condition. The condition can be satisfied when testing against the simulator's memory state alone. In other words, the reply from the dialogue manager does not influence the appropriateness of the responses in the rule.

Result 4: satisfied. The condition is satisfied when testing using both the simulator's memory state and the dialogue manager's reply.

A heuristic score, which can be specified by the developers, is assigned to each of the four results. Lower score means lower probabilities that the templates in the rule will be chosen as the final response.

Response contents

The response contents describe the body of the user response. They are expressed using templates, which means the actual values in the responses are dynamic according to different scenarios and different dialogue states.

```
{c rule
  :conditions ":continuant need_source"
  :response_content (
    {c eform
      :source ":source[:*scenario*]" }
  ) }
```

(a)

```
{c rule
  :conditions ":continuant need_$attr"
  :response_content (
    {c eform
      :$attr ":$attr[:*scenario*]" }
    {c eform
      :$attr ":$attr[:*scenario*]"
      :*other* ":*other*[:*scenario*]" } ) }
```

(b)

Figure 6-4. Examples of response templates.

The templates adopt a very flexible syntax. In Figure 6-4(a), a simple example is shown, which states that, when the dialogue manager asks for a destination in the continuant, the destination should be provided according to the scenario. The string value ":source[:*scenario*]" indicates a reference to the value of the key *:source* in the frame *:*scenario**, where the instantiated scenario is stored. In this case, the key *:source* usually has a single value. If the key referenced has a list value instead, one value from the list is picked by sampling according to the probability distribution. Using the scenario in Figure 6-1, the response template will be instantiated into:

```
{c eform
  :source "BOS" }
```

The same meaning can be expressed using a more generic expression, shown in Figure 6-4 (b). The string starting with a dollar sign denotes a variable, the value of which is the match result from the condition. Therefore, this single rule can provide corresponding information when the system asks for different types of information. In the second response templates of this rule, an additional key *:*other** is also included. This special key can be instantiated randomly into one or multiple elements, among which at least one element has not yet been conveyed to the dialogue manager. Thus, assuming the (simulated) user has only spoken about the destination, the following two responses are both possible instantiations of the second response template.

```
{c eform
  :source "BOS"
  :airline "AA" }
{c eform
  :source "BOS"
  :departure_interval "morning"
  :destination "CHI" }
```

The syntax of the response templates also includes functions like *random()*, which can produce a random number in the given range. For example, the key *:nth* in the following template is declared to take on an integer value between 1 and the value of *:npending*.

```
{c eform
  :nth "random(1, :npending)" }
```

Response command and response action

Each rule may also contain some response commands and a response action. The response command is essentially the frame name of the response. They are made explicit for the purposes of reducing manual effort in composing the rules. A rule with *n* response contents and *m* response commands is equivalent to a rule with *nm* response contents, if the frame names of the responses are specified directly in the templates.

The response action is an action that can possibly change the memory state, including the scenario. For example, in the situation where no suitable flights can be found, the user might want to change his preference. A rule with a response action is shown below. Although the action takes place before the instantiation of the response content procedurally, the action only affects the responses in this rule, and does not have real effect until one of the response templates in the rule is selected as the final response.

```
{c rule
   :conditions ":continuant change_condition"
   :response_action {p change_scenario
            :change ":airline" }
   :response_content ( {c changed
               :airline ":airline[:*scenario*]" } ) }
```

6.1.4 Responses with personalities

The response rules describe the conditions, contents and actions of all the possible responses. Since the condition test only classifies the responses into four classes, in most cases, multiple instantiated responses are scored equally, and thus have equal likelihood to be selected. For a real user, however, the final response is not chosen randomly. Users with different personalities tend to have different behaviors when producing a response. Figure 6-5 exemplifies some possible responses for users with different personalities. We would like to model such differences in the generation of the response.

S: How can I help you? U: I want to go from Boston to Chicago next Tuesday morning by United airline. (a) An aggressive user	S: How can I help you? U: I want to go to Chicago. S: Where are you departing from? U: Boston S: What date are you leaving? U: next Tuesday. (b) A lazy user
S: I have three United flights. They are... U: Book the first one. (c) A cooperative user	S: I have three United flights. They are... U: Show me the fifth one. (d) An uncooperative user

Figure 6-5. Four different types of users. (a) An aggressive user; (b) a lazy user; (c) a cooperative user; (d) an uncooperative user. "S" stands for "systems", and "U" stands for "user".

To start with, we first describe the personality as a multi-dimensional vector, in which the number of dimensions and the underlying meaning of each dimension are determined by the developer. Each dimension represents a personality feature, and takes a value between 0 and 1. Here, we use a two dimensional vector as the example, with the two dimensions representing "aggressiveness" and "cooperativeness" respectively. An aggressive user tends to provide a lot of information in a turn, and make decisions quickly; on the other hand, a less aggressive user behaves more passively, and only speaks the information that is asked for. A cooperative user follows the system's prompts carefully; on the other hand, an uncooperative user might give responses that are not meaningful given the dialogue context.

To simulate a user with a certain personality p^*, the simulator computes a strategy personality p according to the responses chosen so far and the candidate response, and selects with a high probability a candidate response that can minimize the distance between p and p^*. Formally, the probability of choosing response r_i in the k^{th} turn is defined as follows, where z is the normalization coefficient, c is the condition score obtained from the condition test, and $d(\bullet)$ is the Euclidean distance.

$$P(r_i) = \frac{c}{z \cdot d(p_i^k, p^*)} \qquad \text{(6-1)}$$
$$p_i^k = f(p^{k-1}, M(r_i))$$

The strategy personality in the k^{th} turn, p^k, is calculated from that in the last turn, p^{k-1}, and a personality move $M(r)$ results from choosing the particular response r. The personality move is a vector with the same number of dimension as the personality vectors. The integer value in each dimension indicates the number of moves in the specific personality dimension. For instance, $M(r)$=[2, -1] in our example stands for two moves towards aggressiveness, and one move towards uncooperativeness. The value is calculated according to two aspects: the general personality descriptions and the response-specific personality descriptions. The general descriptions, which are a set of logical expressions associated with corresponding personality moves, apply to all responses. For example, for any types of response, if the response contains more than three elements, it is considered as more aggressive, and thus the aggressiveness dimension gains one positive move. The response-specific descriptions are specified in the response templates or the response commands. Each dimension of the personality can have one positive move, no move or one negative move. For example, when the dialogue manager asks to confirm an entity, responding "no" results in a negative move in terms of aggressiveness. The rule is as follows:

```
{c rule
   :conditions ":continuant confirm_entity"
   :response_content ( {c truth
            :truth_value "no"
            :aggressiveness "-" } ) }
```

The total personality move of the response is the sum of its response-specific descriptions and all the moves of the satisfied general descriptions.

The new personality vector $p_i^k = f(p^{k-1}, M(r_i))$ is a nonlinear curve of $M(r_i)$. For each dimension j,

$$f(p,m) = \begin{cases} f(p,m-1) + \dfrac{1 - f(p,m-1)}{s} & m > 0 \\ p & m = 0 \\ f(p,m+1) - \dfrac{f(p,m+1)}{s} & m < 0 \end{cases} \tag{6-2}$$

where s is the j^{th} dimension of a constant vector which defines the size of a move in each dimension. Therefore, the increment is smaller near the two extremes, and the new value always stays in the range of 0 and 1.

The final response is chosen by sampling according to the probability distribution of all the possible responses, and p^k is updated to the new value according to the choice.

6.1.5 Simulation experiments

The simulator was tested with FAUNA in the English flight-reservation domain. The simulator's specification contains three scenario templates corresponding to one-way itineraries, roundtrips, and multi-destination itineraries with two flights. Two personality dimensions, aggressiveness and cooperativeness are defined. Altogether 22 rules are written as the response strategies. After the simulator produces a response in the eform representation, it is converted into an English string using GENESIS with deterministic generation rules, *i.e.* rules and a lexicon without randomness, and sent to FAUNA.

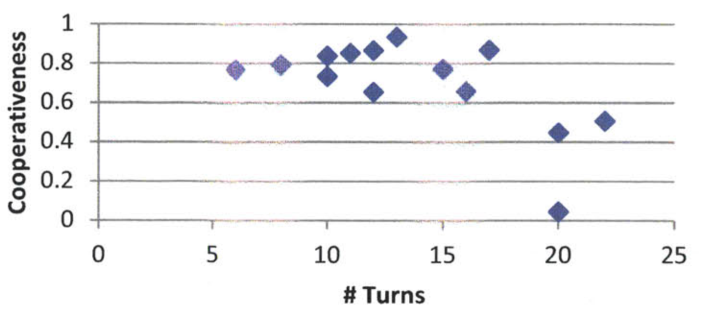


Figure 6-6. Number of turns vs. cooperativeness in the roundtrip scenarios.

U: I'd like to fly to Chicago from Boston

S: flight from Boston to Chicago. What date will you be travelling?

U: one way the friday after February twenty two by any airlines

S: There are 5 flights from Boston to Chicago … Can you be more specific?

U: Show me the latest flight

S: The latest flight is … Do you want to add this into the itinerary?

U: yes

S: Delta flight 164 has been added to your itinerary … Do you want to find out how much your trip will cost?

U: price itinerary

S: the price for the itinerary is 178.0. Is there anything else?

(a)

U: Show me one way flight to Chicago

S: flight to Chicago. What city does the flight depart from?

U: Are there any flights from Boston

S: flight from Boston to Chicago. What date will you be travelling?

U: a flight that leaves on the last friday of February

S: There are 5 flights … Can you be more specific?

U: Choose the seventh flight

S: Sorry, I didn't understand what you said. There are 5 flights … Can you be more specific?

U: I'd like to go to Chicago by United

S: I have a United flight … and a United flight … Would one of these work?

U: Show me that flight

S: I have a United flight … and a United flight … Do you like any of these?

U: I want the earliest flight

S: the earliest United flight is … Do you want to add this into the itinerary?

U: no

S: the next earliest United flight is … Would that work?

U: I want to book that flight

S: United flight 503 has been added to your itinerary … Do you want to find out how much your trip will cost?

U: no

S: Is there something else you'd like to know?

(b)

Figure 6-7. Two examples of simulated dialogues. (a) A more aggressive and more cooperative user. (b) A less aggressive and less cooperative user. "S" stands for "system", and "U" stands for "user".

Fifty dialogues were generated between the simulator and FAUNA. In each dialogue, the simulator simulates a user with a random personality. In the 729 turns of the 50 dialogues, 272 unique sentence patterns were found after replacing the detailed cities, airlines, month names, day names, *etc.*, with corresponding tags. On average, each rule was able to generate 12.4 different sentence patterns, and this number can be further increased when the rules for the language generation system are extended to include more alternative patterns.

To test the effect of the personality in the simulation, we looked at the cooperativeness dimension. Ideally, a cooperative user would have a shorter conversation than an uncooperative user. Figure 6-6 shows the relation between cooperativeness and the number of turns for the most typical roundtrip booking scenario. The number of turns in a dialogue is influenced by the scenario and the availability of the flight. Nevertheless, it can be observed that, when the cooperativeness is below a certain value, the number of turns of a dialogue is generally greater.

Two example dialogues are shown in Figure 6-7. On the left is a more aggressive and more cooperative user. On the right is a less aggressive and less cooperative user. The lengths of the two dialogues are clearly different. We can observe a couple of uncooperative responses (*e.g.* the seventh flight) in the second user, and he appears less decisive than the one on the left.

6.2 Performance assessment

We have discussed dialogue management in the previous chapter, and user simulation in the previous section. The last module indispensable for a dialogue game is the performance assessment. This module gives feedback to the student about how well he is doing. We aim for real-time and comprehensive assessment, *i.e.*, assessment for each turn rather than a pass/fail decision at the end of the dialogue, so that the student can see his performance in the course of the dialogue, and hopefully benefit from both positive and negative assessment to perform better in the rest of the dialogue. Again, syntactic and semantic aspects are our focus, and the pronunciation is not assessed in detail.

The assessment of each utterance in the dialogue context is not easy. The assessment of a whole dialogue might be a lot easier, for at least the system can tell whether the student completes the task or not. Breaking the dialogue down, the utterances in each turn do not have any ground truth answer. The order of providing information differs from student to student, and the sentence structures are changeable. Nevertheless, good utterances should be well-formed by themselves, and contribute to the overall progress of the dialogue. Based on this idea, the turn assessment is designed to include the following three aspects with heuristic scores for each aspect.

Sentence wellness: whether the utterance is grammatical. The grammaticality is judged by the language understanding module, *i.e.* the parser. If the parser is able to produce a full parse and a non-empty eform from the utterance, the utterance is considered as grammatical. In order to encourage more complex sentence structures, this part of the score also involves the sentence complexity, approximated by the length the sentence. A longer grammatical sentence thus receives a higher score than shorter sentences.

Context wellness: whether the utterance fits well in the context. Although the dialogue manager may be flexible enough to handle all sorts of unexpected inputs, as a language learning system, we would like the student to listen carefully to the system's reply and respond accordingly. If the system asks for one type of information, and the student provides another type of information, a deduction is given. For verification and suggestive system replies, however, since the appropriate response can vary a lot in the expression and sometimes the meaning is implicit (e.g. responding to the question "do you want to book this flight" with "are there any other flights"), the assessment is relaxed. The student is also allowed to correct any mistakes made in previous turns without penalty, regardless of the system's reply.

Dialogue progress: how the utterance advances or retrogrades the dialogue progress. The dialogue progress measures how far the current dialogue state has advanced towards a successful conclusion. It is assessed by first extracting key points from the scenario, and computing how many of them have been correctly achieved. The key points include both the elements in the scenario and the entities that are expected to be completed. Every correctly achieved key point is worth one point, while an incorrect key point results in one point deduction. For example, completing one flight is awarded one point, and saying a wrong destination loses one point. The dialogue progress score for each utterance is the difference between the overall dialogue progress after this turn and that of the previous turn.

The scores of the above three aspects add up to a turn score. The assessment of the whole dialogue is the sum of the scores in each turn, plus an independence bonus and a scenario bonus if the student successfully completes the task. The independence bonus is designed to reward students who complete the dialogue on his own. The scenario bonus is proportional to the difficulty and complexity of the scenario.

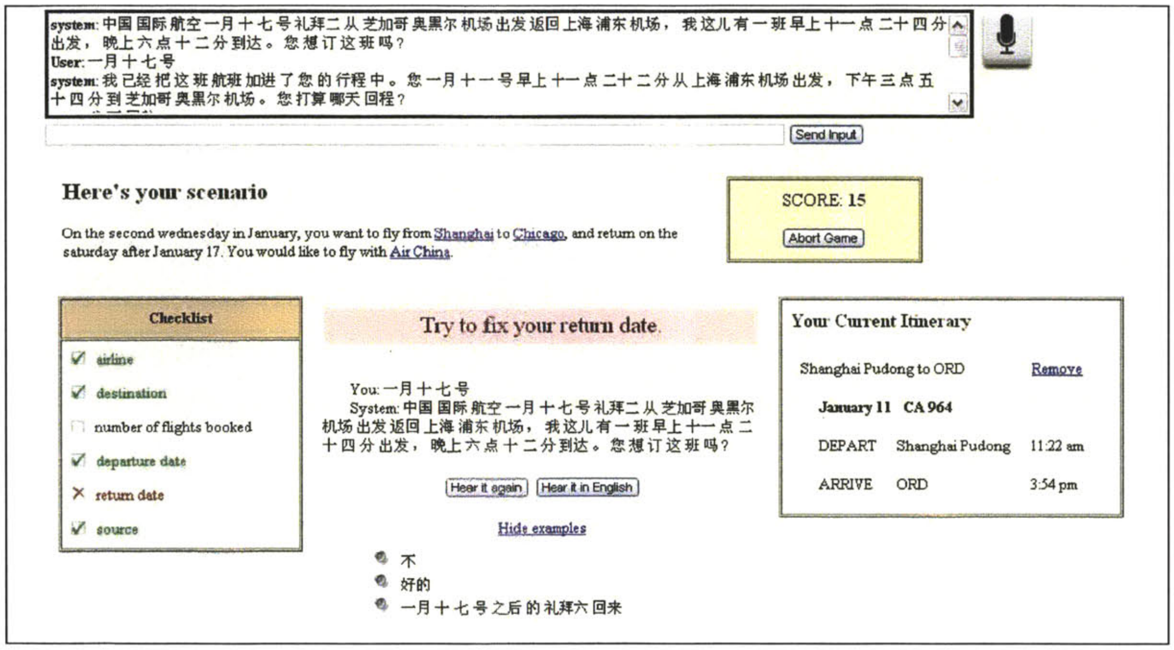
As we can see, the turn score is usually positive, which means a longer dialogue would produce more points than a shorter one. A normalization using the number of turns would work if the scores were to be used to compare two dialogues horizontally. However, we consider it to be more important for the student to speak more than to complete a scenario in a minimum number of turns. Therefore, the score is not normalized. The student earns more points when he speaks more well-formed sentences in the domain. The scores for single dialogues are accumulated across dialogues, and when enough points are collected, the student is advanced to the next level.

6.3 Game implementation

6.3.1 The Mercurial system

So far, all the modules necessary to build a dialogue game for language learning have been discussed. The system's role as a dialogue partner is supported by the dialogue manager, as well as the language understanding and language generation modules. The role as a language tutor is fulfilled by the user simulator and language generator, and the performance assessor. Putting these modules together, a dialogue game for language learning is ready to work.

Figure 6-8 and Figure 6-9 show the screenshots of an implemented system for Chinese learning [71] during a dialogue and after completing a dialogue. The system, implemented in the flight-reservation domain, is named Mercurial after the previous flight reservation system Mercury. During the dialogue, the student sees a scenario expressed in a natural paragraph in the upper area. Below the scenario is a checklist, which gathers the key points from the scenario. As the dialogue proceeds, the items in the checklist are marked with a green check or a red cross according to the correctness of the information conveyed. The current itinerary is shown on the right part.


system: 中国 国际 航空 一月 十 七 号 礼拜二 从 芝加哥 奥黑尔 机场 出发 返回 上海 浦东 机场， 我 这儿 有 一 班 早上 十一 点 二十 四 分 出发， 晚上 六 点 十 二 分 到达。 您 想 订 这 班 吗？
User: 一月 十 七 号
system: 我 已经 把 这 班 航班 加进 了 您 的 行程 中。 您 一月 十一 号 早上 十一 点 二十 二 分 从 上海 浦东 机场 出发， 下午 三 点 五 十 四 分 到 芝加哥 奥黑尔 机场。 您 打算 哪天 回程？

Here's your scenario

On the second wednesday in January, you want to fly from Shanghai to Chicago, and return on the saturday after January 17. You would like to fly with Air China.

SCORE: 15

Checklist

- ✓ airline
- ✓ destination
- ☐ number of flights booked
- ✓ departure date
- ✗ return date
- ✓ source

Try to fix your return date.

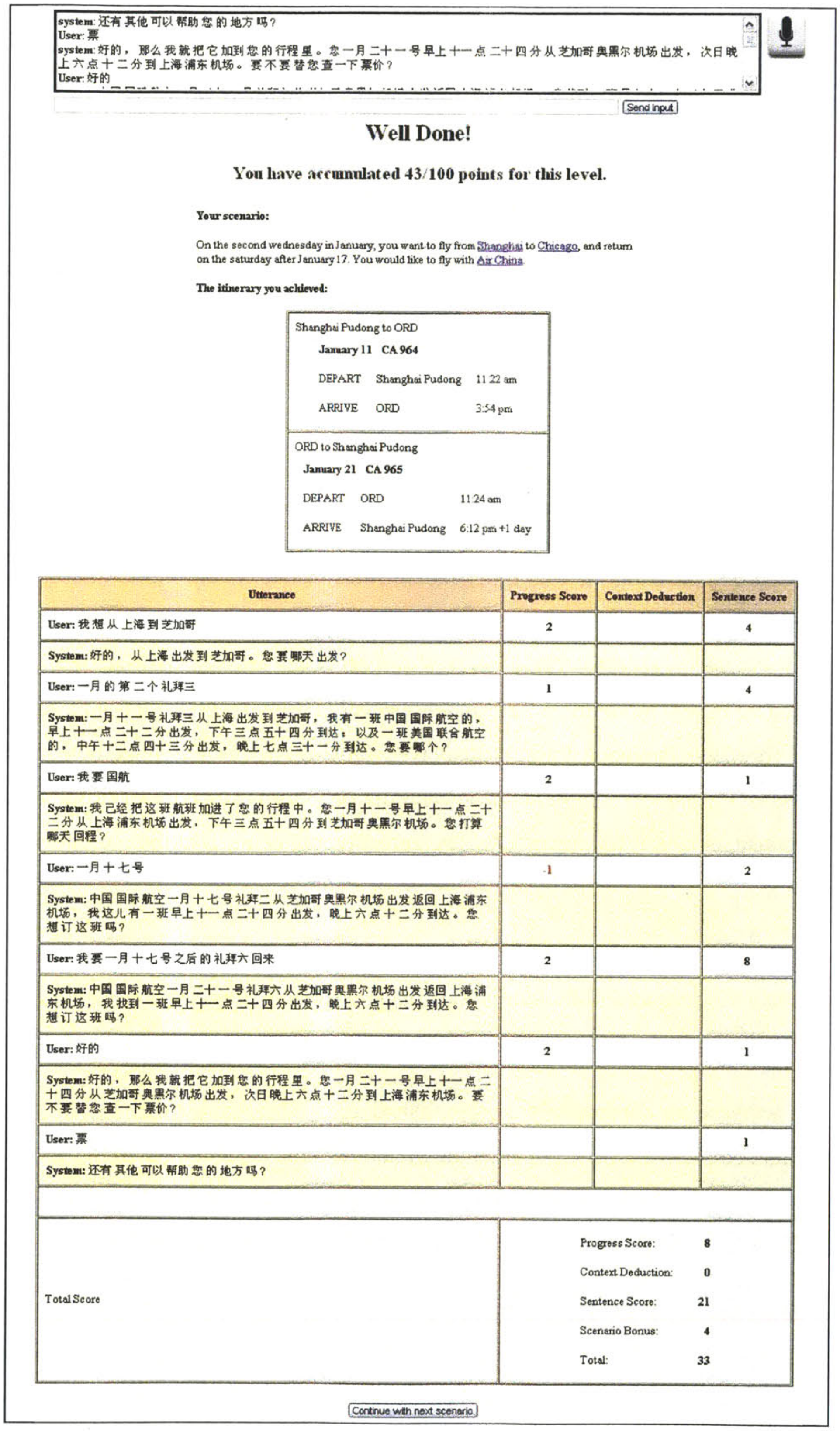
You: 一月 十 七 号
System: 中国 国际 航空 一月 十 七 号 礼拜二 从 芝加哥 奥黑尔 机场 出发 返回 上海 浦东 机场， 我 这儿 有 一 班 早上 十一 点 二 十 四 分 出发， 晚上 六 点 十 二 分 到达。 您 想 订 这 班 吗？

- 不
- 好的
- 一月 十 七 号 之后 的 礼拜六 回来

Your Current Itinerary

Shanghai Pudong to ORD		Remove
January 11	**CA 964**	
DEPART	Shanghai Pudong	11:22 am
ARRIVE	ORD	3:54 pm

Figure 6-8. A screenshot during a dialogue in the Mercurial game. The student has successfully booked the outbound flight according to the scenario, but a mistake was made on the return date. A demonstrative video of the system is available at http://people.csail.mit.edu/seneff/scill/dialogue_game.wmv.



system: 还有 其他 可以 帮助 您 的 地方 吗？
User: 票
system: 好的， 那么 我 就 把 它 加到 您 的 行程 里。 您 一月 二十 一 号 早上 十一 点 二十 四 分 从 芝加哥 奥黑尔 机场 出发， 次日 晚上 六 点 十 二 分 到 上海 浦东 机场。 要 不 要 替 您 查 一下 票价？
User: 好的

Well Done!

You have accumulated 43/100 points for this level.

Your scenario:

On the second wednesday in January, you want to fly from Shanghai to Chicago, and return on the saturday after January 17. You would like to fly with Air China.

The itinerary you achieved:

Shanghai Pudong to ORD		
January 11	**CA 964**	
DEPART	Shanghai Pudong	11:22 am
ARRIVE	ORD	3:54 pm
ORD to Shanghai Pudong		
January 21	**CA 965**	
DEPART	ORD	11:24 am
ARRIVE	Shanghai Pudong	6:12 pm +1 day

Utterance	Progress Score	Context Deduction	Sentence Score
User: 我 想 从 上海 到 芝加哥	2		4
System: 好的， 从 上海 出发 到 芝加哥。 您 要 哪天 出发？			
User: 一月 的 第 二 个 礼拜三	1		4
System: 一月 十 一 号 礼拜三 从 上海 出发 到 芝加哥， 我 有 一 班 中国 国际 航空 的， 早上 十一 点 二十 二 分 出发， 下午 三 点 五十 四 分 到达； 以及 一 班 美国 联合 航空 的， 中午 十二 点 四十 三 分 出发， 晚上 七 点 三十 一 分 到达。 您 要 哪 个？			
User: 我 要 国航	2		1
System: 我 已经 把 这 班 航班 加进 了 您 的 行程 中。 您 一月 十 一 号 早上 十一 点 二十 二 分 从 上海 浦东 机场 出发， 下午 三 点 五十 四 分 到 芝加哥 奥黑尔 机场。 您 打算 哪天 回程？			
User: 一月 十 七 号	-1		2
System: 中国 国际 航空 一月 十 七 号 礼拜二 从 芝加哥 奥黑尔 机场 出发 返回 上海 浦东 机场， 我 这儿 有 一 班 早上 十一 点 二十 四 分 出发， 晚上 六 点 十 二 分 到达。 您 想 订 这 班 吗？			
User: 我 要 一月 十 七 号 之后 的 礼拜六 回来	2		8
System: 中国 国际 航空 一月 二十 一 号 礼拜六 从 芝加哥 奥黑尔 机场 出发 返回 上海 浦东 机场， 我 找到 一 班 早上 十一 点 二十 四 分 出发， 晚上 六 点 十 二 分 到达。 您 想 订 这 班 吗？			
User: 好的	2		1
System: 好的， 那么 我 就 把 它 加到 您 的 行程 里。 您 一月 二十 一 号 早上 十一 点 二十 四 分 从 芝加哥 奥黑尔 机场 出发， 次日 晚上 六 点 十 二 分 到 上海 浦东 机场。 要 不 要 替 您 查 一下 票价？			
User: 票			1
System: 还有 其他 可以 帮助 您 的 地方 吗？			

Total Score	Progress Score: **8**
	Context Deduction: **0**
	Sentence Score: **21**
	Scenario Bonus: **4**
	Total: **33**

Figure 6-9. A screenshot after completing a dialogue in the Mercurial game.

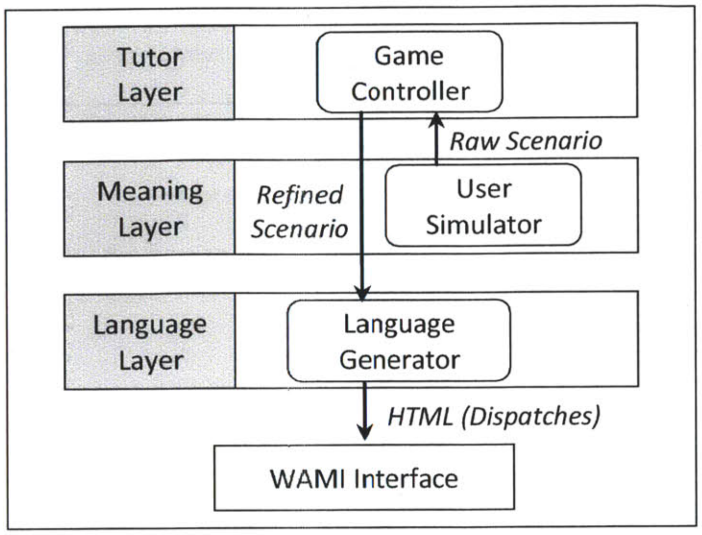


Figure 6-10. System diagram of Mercurial (initialization).

We have been emphasizing the two independent roles of the system in the dialogue game. At the same time, it is not a pleasant experience to be interrupted during the dialogue in order to hear feedback from the language tutor. Therefore, the outputs of the dialogue partner and the language tutor are presented in two modalities. The student hears the reply from the dialogue partner, while the feedback from the language tutor is displayed on the screen. The pink box in the center contains a feedback message commenting on each dialogue turn. The content of the message can be words of praise, or a hint for correction. The current score of the dialogue is shown on the top right corner.

Below the feedback area is the assistance area, where the "repeat" buttons allow the student to hear the system's voice again in the foreign language or in the native language, and the "example" link opens up a list of contextual utterances which the student can say at the current dialogue state.

When the scenario is successfully accomplished or is aborted by the student, the system summarizes the complete dialogue between the system and the student, and shows the student the detailed scores in each scoring aspect. When enough points have been accumulated, the student will be advanced to the next level, in which more complex scenarios are presented.

Looking under the hood, the system diagrams are illustrated in Figure 6-10 and Figure 6-11. At the initialization of each game play, the user simulator instantiates a scenario from the scenario templates. The scenario is reviewed by the game controller and modified to adapt to the current difficulty level. The language generator then receives the scenario, and paraphrases it into a natural paragraph using a random choice of wordings.

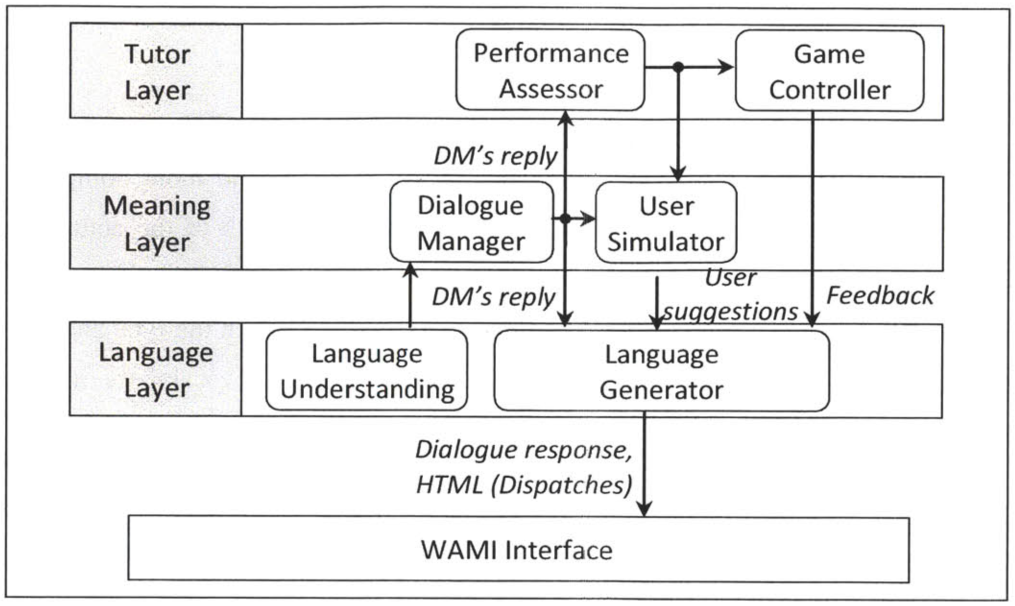


Figure 6-11. System diagram of Mercurial (game play).

During the game play, after language understanding, the meaning representation of the student's utterance is sent to the dialogue manager to produce the dialogue partner's reply. The dialogue manager is fed with a domain specification that is almost the same as the one used in the flight-reservation domain dialogue application discussed in Section 5.6.1, except that the nationality of the knowledge source *user* is changed from "US" to "China". The tutoring mechanism for the statistical engine is turned off, and the two classifiers for task confirmation and focus list (see Section 5.6.1 for details) use the data collected in the experiments of the English version flight-reservation dialogue system.

The reply from the dialogue system is sent to three modules: (1) the language generator to generate a response in natural language; (2) the performance assessor and game controller to produce the language tutoring feedback; and (3) the user simulator to output multiple possible user responses as example utterances. The language tutoring feedback and the possible user responses are then composed into the updated HTML via language generation.

As in the previous three games, TINA is used for language understanding in the two-pass parse setting with a generic Chinese grammar augmented with specialized flight reservation domain vocabulary. GENESIS is used for language generation with alternative rules and lexicon which enable outputs (in Chinese) of different wordings for the same input.

6.3.2 Scenario generation with difficulty levels

Being a game, difficulty levels are an indispensable feature. The idea of the difficulty levels of dialogues is to have easier and shorter dialogues in lower levels, and more complicated and longer dialogues in higher levels. Although the complexity and length of the actual dialogues depend on various factors, the complexity of the scenario is among the most critical ones. Fewer constraints in the scenario usually result in simpler dialogues, and when the type and number of constraints increase, more turns with more sentence patterns and vocabulary are naturally involved.

To generate a scenario that meets the current difficulty level, the game controller takes in the raw scenario produced by the user simulator, and refines the elements in the raw scenario. Each scenario template has a base difficulty level d_{base}, and each element in the scenario is also assigned a difficulty level d_i. The refined scenario is obtained by drawing as many elements as possible from the raw scenario in a random order, such that the difficulty level of the scenario D does not exceed the current difficulty level of the game. D is defined by the following equation.

$$D = \left\lfloor \log_2(2^{d_{base}} + \sum_i 2^{d_i}) \right\rfloor \tag{6-3}$$

From the equation, it can be observed that D is dominated by the highest difficulty level of the elements and the template, but is also affected by the number of elements. A level two element and a level one element result in a level two scenario, and two level two elements result in a level three scenario.

After the elements are selected, the values of the elements are determinized. For elements that takes a probabilistic list as a value, the value with the highest probability is selected. The refined scenario is then sent to the language generator for a natural language paraphrase.

It should be noticed that the generation of the scenario is independent from the dialogue manager, *i.e.*, independent from the flight database. This independency sometimes leads to an unsatisfiable scenario. This is of course solvable by verifying the satisifiability against the dialogue manager in advance, but in Mercurial, it is handled in another interesting way. The unsatisfiable scenario is presented to the student, and if the student interacts with the system properly, the dialogue manager would reply with zero matches at some point. At that time, the game controller notices the zero match, and it then adjusts the scenario by changing or dropping a constraint, and notifies the student of this change via the feedback message. The scenario difficulty is increased, which would add to the student's final score as a bonus if the dialogue concludes successfully. We consider this scenario update feature interesting, because it clearly presents the distinction of the system's two roles: one that helps the student find the flights, and the other that is ignorant of the flight information, but pays attention to the dialogue, notices what is happening, and makes appropriate comments. This also provides greater entertainment value to the student.

6.3.3 Feedback and assistance functions

After the dialogue manager's response is sent to the performance assessor, the assessment result is visualized in two ways in addition to the score: the checklist and the feedback message.

The checklist summarizes the key information in the scenario which is essentially the elements in the refined scenario. All items in the checklist are initially unchecked, and become checked with a green tick if the student conveys the information correctly, or marked with a red cross if the wrong information was communicated. Since the items in the checklist, in other words, the elements in the scenario, are usually the attributes of certain entities defined in the dialogue domain specification, the judgment of correctness is done through calling comparators of corresponding knowledge sources, if available. The checklist gives an intuitive view of the overall dialogue progress. The student is able to know what else they need to say and what to correct by looking at the checklist.

The feedback message, on the other hand, comments on the previous turn. The message includes the following possibilities:

Encouragement: when the student's utterance fails in language understanding.

Context reminder: when the student's utterance fails to follow the system's reply.

Correction hint: when the student provides incorrect information about the itinerary.

Praise on progress: when the student's utterance advances the dialogue progress.

Praise on sentence: when the student's utterance gains a great sentence score.

Assistance is also provided in various ways. Four assistance functions are implemented to offer assistance at different levels. In the comprehension aspect, the system's voice response can be replayed not only in L2, but also in L1 to help understanding. This assistance function is implemented almost at no cost, since the dialogue manager works on the meaning layer, and the language generation rules for L1 English have been developed for the English flight-reservation dialogue system. The last dialogue turn is also displayed on the screen, with Pinyin notations for each word revealed with mouse-over. The student can take the time to read and learn the pronunciation of any new words. This also protects them from getting stuck.

In the composition aspect, the student can ask for assistance at the word level or at the sentence level. For the word level, the key words in the scenario are displayed as a hyperlink. The translation of the word can be seen upon mouse-over, and would be synthesized when clicked. If sentence level assistance is needed, the student can look at and hear the sentences generated by the user simulator. In this game, the user simulator simulates a very cooperative user by setting the condition score of the test result "condition not matched" to 0, so that each response it produces is contextually appropriate and meaningful. The response strategy rules cover both the possible responses at the current dialogue state and possible responses to correct any previous errors. Three responses are chosen randomly from all the possible responses, and paraphrased into L2 using a variety of wordings.

6.4 Improving recognition performance

In the previous sections, various features have been discussed to make the dialogue game plausible. Nevertheless, there is yet another fundamental problem that every speech-enabled system needs to face: the recognition performance. Without a satisfying recognition performance, the merits of the system fade away, and it only makes the student frustrated. Moreover, since this system is designed for language learning, the speech inputs are expected to be highly accented. It is extremely necessary to design certain approaches to optimize the recognition performance.

There are two problems: what to optimize and how to optimize. Word error rate (WER) has been widely used as the most common metric for assessing recognition performance. This is a straightforward metric which compares the recognition hypothesis with the true transcript, and is reasonable to evaluate the recognizer performance regardless of the application. However, considering the performance of the entire system, especially for conversational systems like this, word error rate is not the best metric. WER treats every word as equally important when calculating the figures, which is obviously not true in the context of a dialogue. Errors in many of the stop words are unimportant for language understanding, while misrecognized content words are very likely to cause misunderstanding. In this case, the concept error rate (CER) is more useful than WER. Instead of calculating the error rate in the words, CER calculates the error rate of the concepts contained in the sentence. The concepts are similar to the elements in the eform sent to the dialogue manager, which are exactly the things that need to be correct for proper dialogue progression.

Now comes the second problem: how to optimize the CER. Speech inputs differ from text inputs not only in that they are errorful, but also because the recognizer has the option of sending multiple hypotheses to the understanding components for consideration. Since the user usually expects one single response from the system, the subsequent modules have to perform a multi-to-one mapping to map the N-best list into a single hypothesis. Conventionally, the mapping is a selection process. One common method used in spoken systems is to select the top hypothesis with a threshold. If the confidence score of the top hypothesis is above the threshold, it is passed to the subsequent modules for processing, otherwise, the entire N-best list is rejected. This method relies mainly on the recognizer's capability. Since the recognizer does not have any clue on the meaning of the hypothesis, the rankings of the hypotheses are only based on acoustic scores and language model scores. Selecting the top hypothesis clearly does not necessarily optimize the concept error rate.

Other selection approaches try to defer the decision until subsequent modules have produced useful cues. In the previous Mercury dialogue system [**33**], the decision is made by the parser. The top hypothesis on the N-best list that yields a full parse is selected as the best hypothesis. This ensures that the selected hypothesis is at least grammatical, and thus has a better chance of producing a correct meaning representation. In [**72**], features obtained from the dialogue manager and the domain knowledge are used in addition to the acoustic features, to classify the hypotheses into *accept*, *clarify*, *reject* or *ignore*. The top hypothesis classified as *accept* is selected. If no hypothesis is classified as *accept*, the top one with a *clarify* label is selected, and so on. In their later work [**73**], a statistical user simulator that measures the likelihood that the user would say each hypothesis in the current context was further incorporated. [**74**] adopted a similar approach by using features derived from the recognition score, distributional aspects of the N-best list, and the system's response. But instead of classifying the recognition hypotheses, the classification was optimized based on the system's response.

These works reveal the usefulness of considering syntactic, semantic and discourse information in the decision. In considering the Mercurial system, the situation is slightly unusual in that the speech inputs are highly accented, which makes the acoustic scores more unreliable, but on the other hand, the context is much better known than in conventional dialogue systems. The exact scenario is known at the time of the conversation, which provides strong cues for recognition. To verify this, several N-best selection experiments were conducted. In addition, we also consider a fusion approach to map the N-best list to a single input. Unlike the selection approaches which discard the information in the non-selected hypotheses, the fusion approach fuses the N-best list into a single hypothesis so that information in all hypotheses can be integrated.

6.4.1 Data collection

We first conducted a data collection effort using the Mercurial system for the subsequent experiments. Three native speakers of Chinese and nine learners were recruited to participate in the data collection. Five of the participants contributed to the study voluntarily, and the rest received a gift certificate for their participation.

In order to conduct a meaningful study, we requested the nonnative subjects to have at least two years of Chinese classes or similar experience. They were also asked to self-rank their Chinese proficiency for all four language aspects, shown in Figure 6-12. The nine subjects have a fair distribution in terms of their proficiency. The average scores of the reading, writing, listening and speaking are 2.4, 2.8, 3.0 and 3.0 respectively.

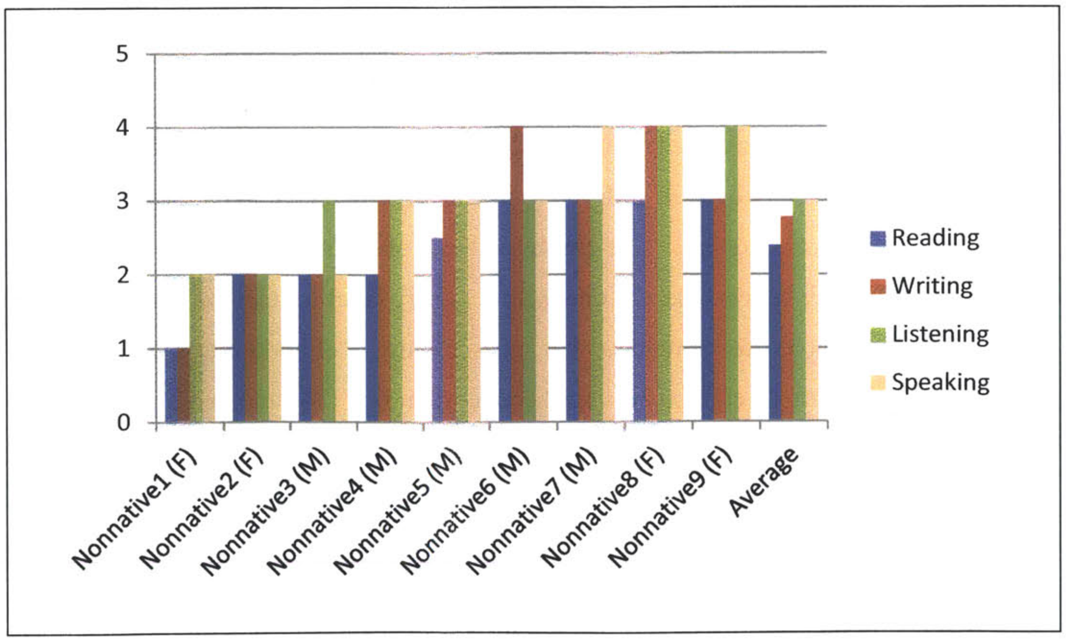


Figure 6-12. Self-ranked Chinese proficiency of the nonnative subjects (1-very poor, 5-native-like). Letters in the parentheses indicate the gender.

Each subject completed 2 to 10 scenarios. The scenarios were randomly generated from several written templates that include different itinerary types (one-way or roundtrip) and different constraints. Each subject started from level 1, and gradually advanced to higher levels according to their performance.

As in the other games, the acoustic models of the SUMMIT speech recognizer were trained from native speech data, to encourage better pronunciation. An *n*-gram language model trained on a flight-domain corpus is used to constrain the recognizer search space. The recognizer outputs 10 best hypotheses for each utterance, ordered by decreasing total score from both the acoustic model and the language model. The language understanding module TINA then produced the eform representation (key-value representation) for each hypothesis. All the hypotheses were sent to the dialogue and performance assessment module.

The utterances collected were manually transcribed and classified into four categories: empty utterances, out-of-domain utterances (*e.g.*, subjects making fun of the system), problematic utterances (ungrammatical, wrong pronunciation, stuttering, *etc.*), and good utterances. The empty utterances and out-of-domain utterances were excluded from the analysis. This left us with 148 native utterances and 509 nonnative utterances. The average lengths of the native utterances and nonnative utterances were 8.7 characters and 7.0 characters respectively.

Figure 6-13 shows the WER and CER of each subject by choosing the top recognizer hypothesis. The WER and CER are calculated using the following equations. The subjects are ordered by their increasing speaking proficiency.

$$WER = \frac{\#word_{ins} + \#word_{del} + \#word_{sub}}{\#word_{ref}} \tag{6-4}$$

$$CER = \frac{\#slots_{ins} + \#slots_{del} + \#slots_{sub}}{\#slots_{ref}} \tag{6-5}$$

For Chinese, the WER is calculated on the character base. The CER was calculated against reference eforms by a human expert. Since the eform key-value representation is hierarchical, it is first normalized and flattened for a more convenient evaluation. Figure 6-14 shows an example of the process. A substitution is counted if both the reference and the hypothesis contain the same key but with different values. An insertion is counted if the hypothesis contains a key that does not appear in the reference. Likewise, a deletion is counted if the reference contains a key that does not show up in the hypothesis.

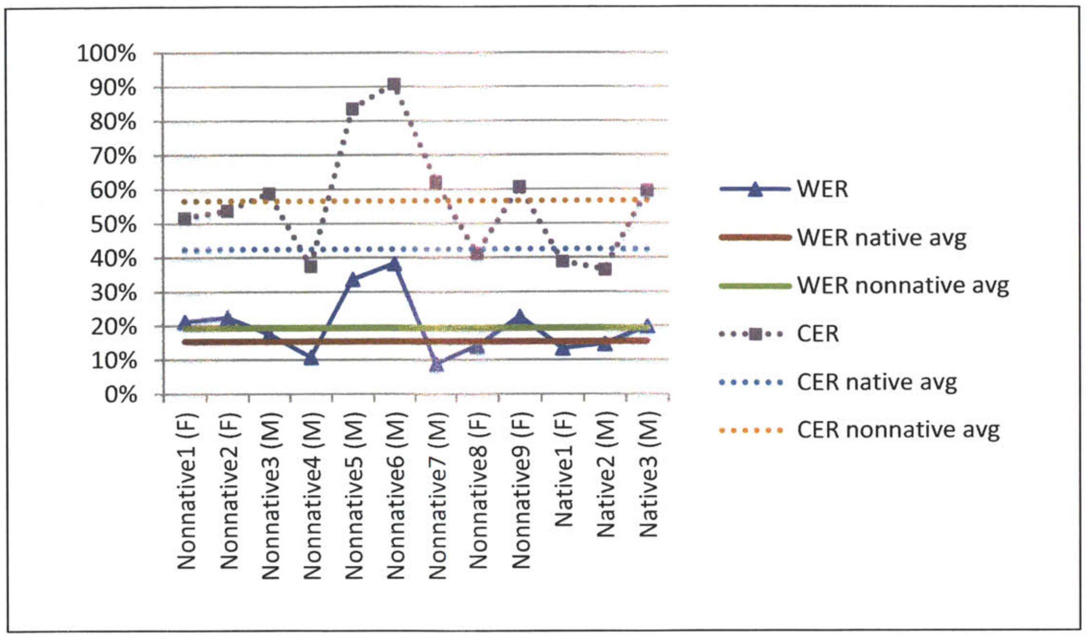


Figure 6-13. Word error rates and concept error rates of the top recognizer hypothesis. The subjects are ordered by increasing speaking proficiency. Letters in the parentheses indicate the gender.

```
{c eform
   :source "NYC"
   :departure_date {c eform
      :month "OCT"
      :day_number 3 } }
```

$\longrightarrow$

```
{c eform
   :source "NYC"
   :departure_date_month "OCT"
   :departure_date_day_number 3 } }
```

Figure 6-14. Normalizing and flattening of an eform.

We can see from Figure 6-13 that the speaking proficiency does not necessarily correlate with the WER or CER. However, native speakers on average had lower WER and CER than the nonnative speakers. The concept error rates are usually much higher than the word error rates because of small denominators and strictness to make a match. For example, if the utterance "I want to go from Boston" is mis-recognized as "I want to go to Boston", the WER is 0.16, but the CER is 2.0 (inserted a destination, deleted a source). It is also noticeable that the CER does not necessarily correlate with the WER either (Nonnative3, Nonnative7 and Native2).

6.4.2 N-best selection

To improve the recognition performance, the first set of experiments adopted the idea of N-best selection. Different cues from different stages of the language processing were incorporated to choose the best hypothesis from the N-best list. Four selection methods were designed and compared.

Top recognizer hypothesis (1-best). The top hypothesis on the N-best list is selected.

Top full parse (parse). The top hypothesis that produces a full parse is selected.

Best dialogue score (dialogue). We first filter out the hypotheses that fail to produce a full parse. For the remaining hypotheses, a dialogue score is computed based on the output of the performance assessor. As described in previous sections, the performance assessor produces scores for an input for four different aspects: sentence wellness, context wellness, dialogue progress and student independency. Since the sentence wellness is implicitly covered by filtering out the non-parsable hypotheses, and the selection task has nothing to do with the independency, the dialogue score used for N-best selection is the sum of the dialogue progress score and a variant of the context wellness score. The context score is not directly used because it only distinguishes inputs which are absolutely wrong from those that are okay. Among the latter, the likelihood is not the same for all inputs. For example, responding to "do you want to book this flight" with "are there any other flights" is okay, but responses like "yes" and "no" are more likely. The variant of the context score distinguishes three situations: "wrong context" with a score of -1, "highly expected" with a score of 1, and "neutral" with a score of 0.

Combined score (combined). The combined score combines the dialogue score and the information from the recognizer. In the first phase of data collection, the acoustic scores were not logged; therefore we assign heuristic N-best rank scores to the hypotheses. The top three hypotheses receive three points, the next three receive two, and the rest receive one.

Table 6-1 lists the WER and CER of the four N-best selection methods. It is clear from the table that using the dialogue score helps both the WER and the CER. Statistically significant improvements were obtained in terms of both WER and CER for both native and nonnative when the dialogue scores and the N-best rank scores were both incorporated. Especially for the nonnative CER, over 12% absolute improvement was gained. We also experimented with real scores from the recognizer (the acoustic score plus the language model score) for the combined method. The results showed no significant difference from using the N-best rank scores.

Table 6-1. WER and CER of the N-best selection methods. Bold indicates statistically significant improvement over 1-best method.

	Native WER	Nonnative WER	Native CER	Nonnative CER
1-best	15.3%	19.3%	42.5%	56.5%
parse	15.3%	19.0%	41.4%	56.3%
dialogue	14.0%	**17.3%**	**35.0%**	**45.0%**
combined	**13.9%**	**17.2%**	**33.3%**	**43.7%**

6.4.3 N-best fusion

6.4.3.1 Oracle experiments

The N-best selection experiments verified that using context cues to select the best hypothesis improves the CER. However, selecting a single best hypothesis ignores the information contained in the rest of the N-best list. It is conceivable that fusing the N-best list by selecting information most likely to be correct in different hypotheses into a single hypothesis would further improve the performance.

Since we are more concerned with correctly understanding the user's meaning, the approach explored here fuses the N-best list at the level of eform representations, *i.e.*, selecting appropriate key-value pairs from the entire N-best eforms to form a final result. With the eform representations, the unimportant information, such as carrier words, has already been discarded during language understanding, resulting in fewer distractions for the fusing process.

As a validation of the feasibility of this idea, an oracle experiment was first conducted. The oracle works as follows: for every key-value pair in the reference, if it exists in one of the N-best eforms, the algorithm adds it into the final fused result. Therefore, the oracle algorithm does not produce any substitution or insertion errors. All the possible errors are deletion errors.

Table 6-2 shows the CER of the oracle algorithm, in comparison with the N-best selection oracles that optimize the WER and CER respectively. WER is not calculated, since the fusion algorithm might produce a result that is not in the original N-best list, and it would be challenging to rebuild the utterance from a fused eform representation.

Table 6-2. CER of different oracle algorithms for N-best selection and N-best fusion.

	Native	Nonnative
Selection (WER)	27.8%	33.9%
Selection (CER)	16.9%	23.8%
Fusion	11.9%	16.9%

The fusion oracle substantially outperforms both selection oracles in terms of the CER, which is promising for exploring real fusion methods.

6.4.3.2 Heuristic Fusion

To fuse the N-best eforms into one, appropriate key-value pairs need to be selected from the N-best candidates. Observing a bit more closely, it can be noticed that the keys and the values in the eforms represent different types of information. The keys are usually derived from syntactic structure, while the values usually correspond to content words. For example, the keys *:source* and *:destination* are derived from two prepositional phrases, and their values are the objects of the prepositional phrases. Thus, the keys should be more robust than the values, because the vocabulary to form the syntactic structures is much smaller than that of the content words, and usually is well covered in the language model. To take advantage of this property, we would like to separate the tasks of selecting the keys and selecting their values.

On the other hand, the dialogue scores obtained from the system are attributed to the key-value pairs, not the keys alone or the values alone. Therefore, the final algorithm we came up with scores both the key-value pair as a whole, and the keys and values separately.

Key-value pair scoring

The key-value pairs are scored according to their contribution towards the dialogue scores. However, it is not easy to obtain the dialogue score for each key-value pair for two reasons. First, due to the uncertainty embedded into the system, it is very hard to reproduce exactly the same scenario and the same dialogue. Secondly, certain dialogue progress is credited toward a combination of multiple key-value pairs, rather than a single pair. For example, the combination of a correct month and a correct day number leads to one credit in the dialogue score. Therefore, instead of trying to obtain the dialogue score for every single key-value pair, the correlation between its occurrence in the N-best list and the dialogue score of each hypothesis is calculated. The detailed formulation is as follows, where *C* is the occurrence vector, and *D* is the dialogue score vector for the N-best hypothesis. If the key-value pair exists in the i^{th} hypothesis, its occurrence c_i is assigned 1. If the key appears in the hypothesis but with a different value, the occurrence is assigned -1. Otherwise, zero is assigned to c_i.

$$s_{kv}(k,v) = corr(C,D)$$
$$c_i = \begin{cases} 1 & (k,v) \in hyp_i \\ -1 & (k,v') \in hyp_i, v' \neq v \\ 0 & otherwise \end{cases} \tag{6-6}$$

Key and value scoring

Each key (or value) is scored by its weighted count. The equation is shown in (6-7), where w_i is the weight for each N-best hypothesis calculated using the dialogue score and the N-best rank score discussed in the previous subsection.

$$s_k(k) = \sum_i \delta(k,i) w_i$$
$$\delta(k,i) = \begin{cases} 1 & k\ appears\ in\ hyp_i \\ 0 & otherwise \end{cases} \tag{6-7}$$

Selection

To produce a fused eform, the key-value pairs that either have high contribution to the dialogue score or have a high weighted count are selected. Formally, the pair *(k,v)* is selected if it satisfies either of the following two criteria.

(1) $s_{kv}(k,v) > thres_{kv}$

(2) $s_v(k) > thres_k \sum_i w_i$, and for any other possible values *v'* of *k*, $s_v(v) \geq s_v(v')$

Notice that the hierarchy of the eform is not taken into consideration for the fusion process. The selection of the key-value pair only works for the leaf key-value pairs. To re-create the hierarchy, the most frequent parent, if any, is assigned to the pairs.

6.4.3.3 Fusion with SVM

Another approach we experimented with is fusion using an support vector machine (SVM) classifier [**75**]. The classifier classifies each key-value pair into POSITIVE or NEGATIVE, and the POSITIVE pairs are selected to form the fused eform. If multiple pairs with the same key are classified into POSITIVE, the one with a higher confidence score is retained. Table 6-3 lists the features used for classification.

Table 6-3. Features used in the SVM classifier for N-best fusion

Feature	For the pair (k, v)	For the key k
Percentage of occurrence in the N-best list	√	√
Index of first occurrence in the N-best list	√	√
Sum of dialogue scores of the hypotheses it appears in	√	√
Sum of recognizer rank scores of the hypotheses it appears in	√	√
Correlation with the dialogue scores	√	

Table 6-4. CER of the fusion methods. Bold shows the statistically significant results against the selection method ($p < 0.01$).

	Native	Nonnative
Selection (combined)	33.3%	43.7%
SVM Fusion (5 features)	31.7%	44.0%
SVM Fusion (9 features)	32.8%	47.6%
Heuristic Fusion	32.0%	**40.2%**
Manual Fusion	**27.2%**	**34.3%**

6.4.3.4 Results

Table 6-4 shows the CER result of the two fusion methods in comparison with the N-best selection method. For heuristic fusion, we chose $thres_{kv}$=0.8 and $thres_{k}$=0.6. We used a linear kernel For the SVM experiments. Due to the small amount of data we have, the SVM fusion results were obtained via leave-one-speaker-out cross validation. We trained the classifiers with all the nine features mentioned above, as well as with a reduced feature set which only contains the five features for the pair (k, v). The classifiers with the reduced feature set gained better results than the ones with the full feature set, probably because the amount of data was not sufficient for the additional features. The heuristic fusion method gained statistically significant improvements on the nonnative data. For the native data, the CER was lowest using the SVM fusion, but the result was not statistically significant.

Compared to the oracle results, the figures reveal a considerable gap. However, it should be noted that the oracle was optimized for the CER. In the case of a misrecognition where all the hypotheses are wrong, the oracle produces a result that minimizes the CER regardless of the frequencies and all other observable features of the key-value pairs. To make a fairer comparison, we also did a manual fusion experiment with an expert. The human did not have access to the true transcript, and produced a fusion result only by looking at the N-best eforms, as well as the dialogue scores. The result showed that the human still performs better than both methods. One significant difference we noticed is that the human takes into account the existence of other keys when deciding whether a particular key should be selected or not. This is not modeled by either of the methods, and can be a potential topic for future work.

6.5 User study

A subjective evaluation was conducted by the subjects participating in the data collection. A questionnaire was given to each subject. The questionnaire, shown in Figure 6-15, contained seven questions asking about the interest level, difficulty and helpfulness of the game. Various features are also assessed in terms of their helpfulness. We also asked the subjects about their previous experience with computer-based Chinese-learning software. Four out of the nine nonnative subjects responded that they had used such software such as *Rosetta Stone*, and *Fluenz*.

The subjects responded positively about the game. Figure 6-16 and Figure 6-17 show the average scores of the questions in the questionnaire. The average scores for the interest value and helpfulness of the game are both 4.3. The aspects where the subjects considered the game helped most are speaking, sentence patterns, vocabulary and reading. Some other aspects that the subjects mentioned in which the game is helpful are discourse and practical application. All subjects would recommend the game to other learners of Chinese. The top five popular features were the checklist, the level advancement, the example utterances, the feedback message and the scores.

1. Was the game interesting to you? (1 – Not interesting at all; 5 – Very interesting) ___
2. Was the game difficult to you? (1 – Very easy; 5 – Very difficult) ___
3. Was the game helpful in learning Chinese to you? Or if you're a native speaker, do you think it would be helpful for students learning Chinese? (1 – No help at all; 5 – Very helpful) ___
4. In which aspects do you think the game is most helpful? (Multiple selections)
   a. Vocabulary b. Sentence patterns c. Reading
   d. Speaking e. Listening f. Other ________________
5. Did you find the following features helpful/useful?
   (1 – I didn't notice this feature at all; 5 – It's very helpful/useful)
   a. The feedback message in the center __________
   b. The example sentences in text __________
   c. The example sentences in speech __________
   d. "Repeat" in Chinese __________
   e. "Repeat" in English __________
   f. The checklist on the left __________
   g. The scores you get __________
   h. Level advancement __________
   i. Different scenarios within each level __________
   j. Different scenarios across levels __________
   k. Dialogue summary at the end
6. Would you recommend the game to other people who are learning Chinese?
   a. Yes b. Not sure c. No
7. Any other comments you want to share?

Figure 6-15. The subjective questionnaire given to the subjects.

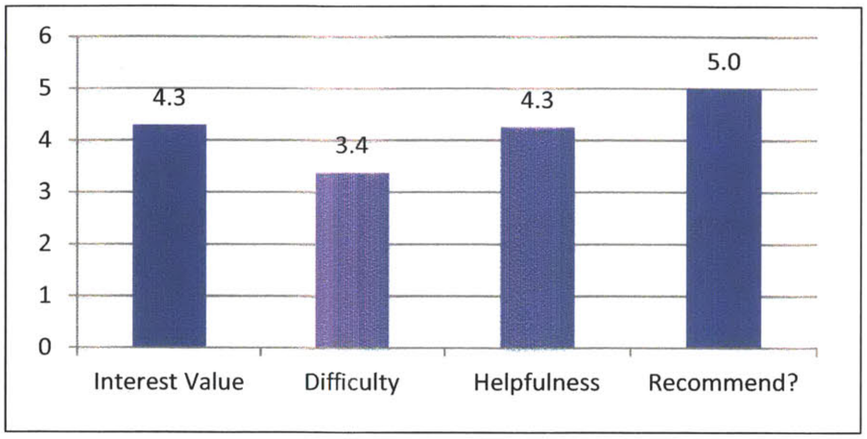


Figure 6-16. Average scores of Question 1 (interest value), Question 2 (difficulty), Question 3 (helpfulness), and Question 6 (recommendation).

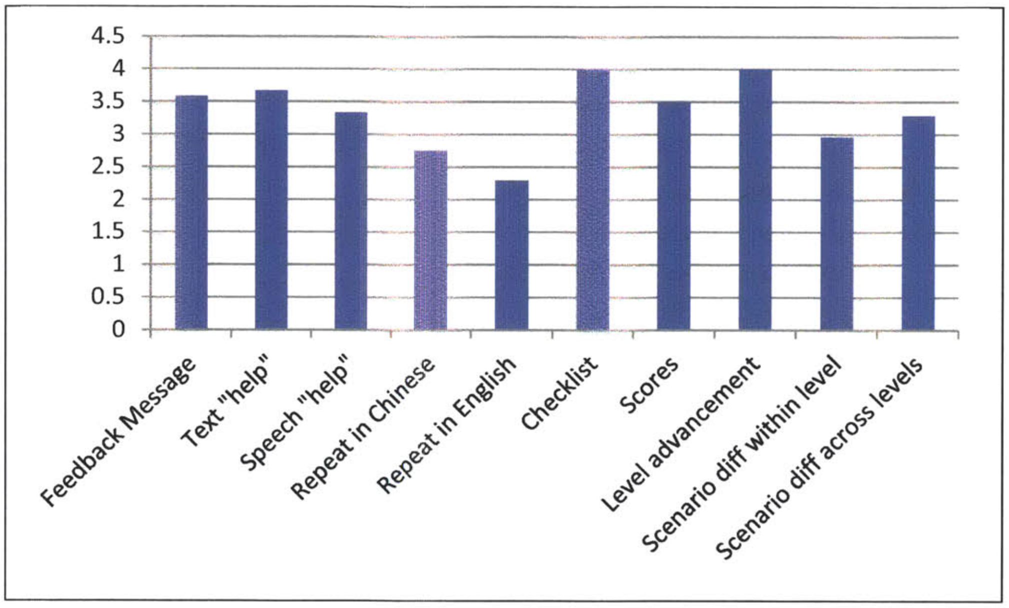


Figure 6-17. Average scores of helpfulness of various tutoring features.

We are also interested in how subjects with different Chinese proficiency responded differently. We classify the nine nonnative subjects into two groups according to the average scores of their self-ranked proficiency of the four language aspects. Subjects with an average score lower than 3 are classified into the lower proficiency group, and subjects with an average score higher than 3 are classified into the higher proficiency group. The average scores of the questions of the two groups are shown in Figure 6-18. The lower proficiency group considered the game to be more interesting and more helpful. Most of the tutoring features appeared to be more helpful to the lower proficiency group, especially the feedback message and the repeat in English. However, synthesizing the example utterance seems to be more attractive to the higher proficiency group. This could be due to the speaking rate of the synthesized voice. Some of the nonnative subjects with lower proficiency actually complained about the synthesized voice speaking too fast, so they would rather read the text on the screen than listen to the speech.

Some subjects were amazed by the level of automation and intelligence of the system. One subject commented that there are only four or five minutes for one person to speak in one Chinese class, but with this automated system, he can practice speaking a lot more. Another subject asked whether the system would be made available soon to the public. Other subjects suggested that we should extend the system to other domains, as well as slow down the synthesized voice for them to practice listening more effectively.

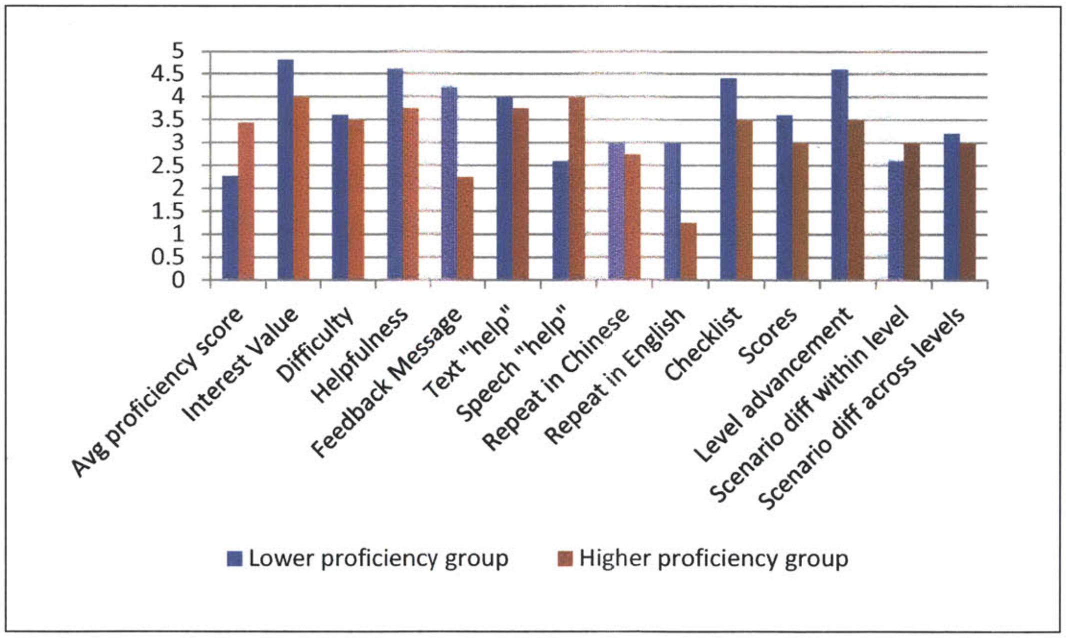


Figure 6-18. Average scores of Question 1 (interest value), Question 2 (difficulty), Question 3 (helpfulness) and sub-items in Question 5 (helpfulness of the features) by the two nonnative proficiency group.

6.6 Summary

Dialogue is a comprehensive and challenging activity for language learning. Unlike the previous three games in which the content of the utterance to speak is given in either L1 or L2, in a dialogue activity, the student has to comprehend the dialogue partner's speech, and based on that, compose a spontaneous response. Due to the complexity of handling the dialogues, dialogue exercises in existing language learning software are dominated by scripted dialogues or semi-scripted dialogues.

This chapter presented a dialogue game with full-functional natural language processing. The role of the system is divided into two parts. It converses with the student as a dialogue partner, and it provides assessment and feedback to the student as a language tutor. The first role makes it a dialogue system, and the second role distinguishes it from conventional dialogue systems designed for native speakers. The separation of the roles also makes it possible to develop more generic modules that can be used not only in language learning systems, but also in other general-purpose systems.

In order to handle the dialogue, a new dialogue manager FAUNA for goal-directed dialogues has been introduced in the previous chapter. In short, FAUNA handles the dialogue in a problem solving manner according to a declarative entity and knowledge source specification. A statistical classification engine is embedded in FAUNA so that statistical decision making can be incorporated. FAUNA has been used in two English dialogue systems for native speakers, as well as in the flight-reservation domain dialogue game Mercurial for Chinese learning. These three applications demonstrate its language independency and domain portability.

In parallel with FAUNA, a user simulator is also developed. The user simulator creates the user intention based on the scenario templates. The responses are generated according to the strategy rules, which encode the conditions and response templates. User personality is also modeled, so that different responses are assigned different probabilities when simulating users with different personalities. The user simulator can be used to simulate dialogues in conjunction with FAUNA, or to provide user suggestions without affecting the real dialogue. The two ways have been tested in an English domain and a Chinese language learning environment respectively.

The whole framework for the dialogue game is completed with a control module and an assessment module. The control module controls the level advancement and revises the scenario according to the current difficulty level. The assessment module evaluates the student's performance on the basis of five aspects: sentence wellness, context wellness, dialogue progress, independence, and scenario complexity.

With all the modules developed, a real dialogue game in the flight-reservation domain, Mercurial, is presented for Chinese learning. This system is the most sophisticated and full-function dialogue system for language learning in the field as far as I know. The student talks with the system in fully natural language, receives feedback from a checklist, feedback messages, and a score, and has access to assistance contextually and bilingually.

The system was evaluated by real subjects. Three native subjects and nine nonnative subjects were recruited to interact with the system. Based on the data collected, experiments to improve the recognition performance were conducted. We compared four N-best selection methods and two N-best fusion methods, and concluded that using the dialogue context cues can significantly improve the recognition performance, especially under the circumstances where the nonnative speech is highly accented.

The system received positive feedback from the subjects participating in the evaluation. Statistics from the questionnaire showed that the subjects consider the system interesting and helpful. The system is especially helpful in practicing their speaking abilities. The subjects would recommend the system to their friends, and they would like to see the system being extended to other domains and being available to the public.

Chapter 7 Summary and Future Directions

This thesis has investigated the area of computer-assisted language learning, particularly using language technologies to create activities that exercise the students' comprehension, composition and speaking abilities. A series of four language learning games, reading, translation, question-answering and dialogue, featuring semi-automatic content creation, automatic assessment and automatic assistance generation, was discussed and implemented. This chapter will summarize the major contributions of the thesis, and point out possible future research.

7.1 Contributions

The topics of this thesis center around systems for computer-assisted language learning, and cover a wide space in the area of spoken language processing. The methods we explored and developed were aimed to be generic, while at the same time, certain amounts of pragmatics were included to enable building real systems.

The contributions of this thesis can be organized into three main aspects: the framework, the modules, and the game systems.

7.1.1 Framework

To build any language learning system, whatever the activity is, several functionalities must be implemented. In Chapter 3, we discussed the responsibilities of traditional language teachers, and, derived from that, summarized three key functionalities of a computer system for language learning: content creation, assessment, and assistance. These three functionalities involve different language processing technologies, and may also differ for different activities. In order to allow maximum sharing of the various modules among different activities, a three-layer conceptualization is proposed which clearly defines the language- and domain- independency of the modules on each layer: the language layer serves as the natural language interface in a specific language, the meaning layer handles the semantics with a language-independent property, and the language tutor layer controls the activity procedure and provides application-dependent tutor features, including assessment. The concept is implemented using the DCTL framework, and the entire architecture is completed with the WAMI toolkit to support a web-based user interface and communication with recognizers and synthesizers. By arranging the modules differently via the DCTL script, different applications can be built with different levels of language processing. This also serves as a basic architecture for all kinds of WAMI-based systems that involve heavy natural language processing. Many of the sub-DCTL files written for the language learning systems, such as producing synthesis/HTML dispatches, handling a dialogue turn, and handling multilingual inputs from the input box, are generic, and can be utilized by other systems, not necessarily involving language learning.

7.1.2 Generic modules for spoken language processing

In order to create exercise contents and to understand and process the student's input, a number of language processing modules that cover a wide space of language processing technologies were developed. These modules were developed in a generic way so that applications other than language learning systems may also use them as a part of the whole system.

In Chapter 4, a template-based content generation method was described. The method reduces human effort in providing domain-dependent contents. The contents can be generated in monolingual or bilingual form. Blending and balancing models enforce a good coverage of both new and review materials. The method can be used to generate corpora other than lesson materials as well, for example a corpus for language model training and system testing.

In Section 5.3.2, a transformation approach was proposed to automatically pose questions from statements. The approach works from the meaning level, and can be applied not only to statement-question transformation, but also other to types of meaning/structural modification of a given sentence. Section 5.3.3 proposed a simplified context resolution method via alignment to help judge the correctness of an answer to a question. The method can be applied in other question-answer situations to expand the abbreviated answers into full context.

In Chapter 6, a brand new dialogue management model was proposed and implemented. The dialogue manager uses the entity goals and knowledge source constraints in a declarative domain specification to plan the dialogue progress as a problem-solving procedure. Statistical classifiers supplement the entity-constraint-based management to handle decisions not easily describable by rules. The tutoring mechanism built inside the dialogue manager allows the developers to teach the system appropriate behaviors by interacting with it and correcting it when the statistical engine makes a mistake. The dialogue manager has been successfully used to implement dialogue systems in two distinct domains and two distinct languages to demonstrate its domain portability and language independence.

This new dialogue manager provides a new model for handling goal-directed dialogues. Developers are able to build dialogue systems in a specific domain in a reasonably short period of time and with a small amount of domain-dependent code. The combination of reasoning and statistics is also a novelty that allows the developers to have good control of the system behavior, while still making use of the statistical power. The novel tutoring mechanism for building the models enables the system to be developed without any pre-existing data. The mechanism also sets the foundation for future system personalization through the interaction between the end-user and the system.

In Section 7.1, a user simulator was described that models the personality of the user. A specification file guides the instantiation of user intention, and the user responses are chosen based on both the current dialogue state and the specified personality of the simulated user. In experiments with a two-dimensional personality feature vector, obvious differences could be observed between a less cooperative and less aggressive user and a more cooperative and more aggressive user. This simulator was used to generate user suggestions in the dialogue game. It can also be used for generating dialogue corpora and testing a dialogue system. The novel personality model makes it especially suitable for stress-testing the dialogue systems by simulating "ill-behaved" users.

Other minor contributions in terms of the language processing modules include improvement to the parsing and language generation performance (Section 5.2.2), and detection of semantic contradiction in a set of statements (Section 5.3.4), *etc.* All of these modules were designed in a generic fashion to serve general purposes. Various applications can be developed with the help of these modules.

7.1.3 Four games

Four real web accessible games for Chinese learning were implemented ranging from reading and translation to question-answering and interactive dialogue with increasing difficulty. In the reading game (Section 5.1), the student sees a list of randomly generated Chinese sentences, and is asked to read them out loud in any order. In the translation game (Section 5.2), the system shows a list of English sentences, and asks the student to compose and speak Chinese sentences with equivalent meanings. In the question-answering game (Section 5.3), a list of Chinese statements is displayed. The student listens to the questions spoken by the system and provides spoken answers based on the statements. Finally in the dialogue game (Chapter 7), the student converses with the system naturally to fulfill a task given by the system. Various kinds of assistance and hints are provided in these games. Assessment is given for each utterance, as well as for a longer period of game play in the form of game points, which allow the student to advance to the next level when enough points have been collected. These systems are very unique in the field in that they focus on the student's comprehension and composition abilities, and process the student's speech to understand the meaning and give feedback. Students are thus allowed to interact with the system using natural and flexible speech, with access to versatile hints and rich feedback.

The games were evaluated using real subjects. The overall system performance was satisfying, and the subjects considered the systems as interesting and helpful. Using the data collected, several experiments were conducted to improve the recognition performance via the N-best hypothesis selection and N-best meaning fusion. Experimental results showed that including context cues helps reduce the concept error rate of the recognition outputs, and N-best fusion was able to achieve a lower concept error rate than N-best selection.

These games demonstrated feasibility of building complex systems for language learning. The model they created can be borrowed to develop similar systems, or be extended for other types of activities. Besides, these four games also provide a platform for future user studies on computer-assisted language learning. Researchers can utilize the systems to create different conditions and analyze the user behavior.

7.2 Future directions

Many aspects of the work in this thesis can be extended. Five possible future directions are described below.

7.2.1 Larger-scale and customizable learning materials

Given the successful implementation of the games in the flight and travel domain, one natural extension is to develop the systems in other domains. Comparatively speaking, the first three games are easier to port to other domains than the dialogue game. We would like to have language teachers to be involved and prepare lesson materials in larger scale to create games that can be put into real use. An attempt has been made to create a question-answering game for a video clip obtained from a summer Chinese language learning camp. The content of the video clip was manually summarized to make up a lesson template, and the system was set up successfully in two days with only minor grammar revisions mainly to incorporate more vocabulary to cover the content. This demonstrated the portability of the system, as well as the potential to support learning materials in more languages. In fact, the model of the translation game has been adopted by other researchers to create multi-lingual translation games [**76**][**77**].

Customizable learning material is another direction. In traditional language learning, the materials are created by the teacher. A web-based CALL system like this, however, can be a tradition-breaker. The student can possibly be a content creator to compose lessons that have particular interest value to himself. An example has been shown in the Word War game [**42**] for vocabulary learning in which the student can create and share their own sets of flash cards. In games for learning at the sentence level, it is more challenging to have the student be the content creator. But, as mentioned in this thesis, the lesson templates for the reading and translation games were written in L1. This implies that the student can possibly use their native language to create lesson materials, and have the computer translate the materials and generate proper exercises. An interactive interface can be developed to allow people without any training to compose lesson templates. With automatic grammar and language model training, students would be able to create and learn using their own personalized lessons instantly.

7.2.2 Paragraph reading comprehension

The question-answering game can possibly be extended into a full reading comprehension exercise. The idea is the same: the system shows the student some texts in L2, and asks questions based on the texts. In the current version of the question-answering game, the texts are isolated statements. It would be more interesting and challenging for the students if the contents were coherent stories or paragraphs. There are two possible ways to realize the idea: generating coherent paragraphs and using existing texts from various sources. Both ways impose challenges. Generating paragraphs requires techniques to produce coherent and natural sentence sequences with appropriate ellipsis and coreference, while still maintaining the variety of the generation outputs and low human effort required to compose the materials. On the other hand, using existing texts requires mechanisms to analysis the texts, resolve the ellipsis and coreferences, and summarize the main idea, so as to be able to generate questions both on the sentence level and on the higher paragraph level.

7.2.3 More dialogue activities

This thesis demonstrated a dialogue game in the flight domain where the student is asked to conduct a dialogue with the system to achieve a flight itinerary which meets the given scenario. Using the dialogue manager and other modules developed in this thesis, more dialogue activities can be created in terms of both domains and forms. A similar domain to the flight reservation domain could be ordering at a restaurant, where the student is given the dietary and budget constraints, and is asked to order from a menu, satisfying the constraints.

The dialogue activity can also be carried out in a different way, in which the system as the language tutor does not give feedback after each turn; instead, the student is asked to answer some questions after the conversation. For example, the student interacts with the system about the weather tomorrow. Afterwards, he is required to answer questions such as the temperature or the likelihood of precipitation. The questions may be in the written form, or more challengingly, also in the spoken form. Compared to the one implemented in this thesis, such a dialogue system would have a more sophisticated design, in which the computer plays three roles: the dialogue partner (information provider), the language tutor, and the information seeker. An even more complicated dialogue activity, however, can be imagined by combining multiple dialogue activities. For example, in the first scenario, the student talks with the system, which acts as his friend, to set up a dinner schedule. Then in the second scenario, the student interacts with the system to find a suitable restaurant and make a reservation. In the final scenario, he orders the dishes according to the given preference and budget.

The discussion above is all centered on goal-directed dialogues. Another direction of the dialogue activity is goal-less dialogues, in other words, chatting-style dialogues. One way is to use data-driven methods, such as the "chatbots" for instant messaging services on the Internet. However, these chatbots are mainly designed to amuse users. Because they do not distinguish valid and invalid inputs and they are not guaranteed to produce meaningful responses, they are not suitable in the language learning setting. In the work of [**78**], a framework for building chatting-style dialogues for language learning was proposed. The idea originated from the casual chat between a shopkeeper and a customer in the free market, where a friendly banter might lead to a better bargain. The system was able to produce dialogues using a template-like structure to track the customer model and the shopkeeper model. Nevertheless, it generally still remains challenging to handle dialogues reliably where there is no particular goal, and thus the scope of the input utterances is too large to model.

7.2.4 CALL system evaluation

Another interesting topic to explore is the evaluation of computer-assisted language learning systems, especially for complex systems. Objective evaluation can be performed for some system components, for example, the word error rate and concept error rate of the speech recognizer, or the percentage of appropriate system responses. However, these measurements do not reflect the entire performance of the system. As the main purpose of a language learning system is to help the student improve language proficiency, the pedagogical effectiveness and user satisfaction are major concerns. These, however, are very hard to measure quantitatively. Most work relies on questionnaires to gather users' opinions, such as [**79**] and [**49**]. In [**49**], a pre-test disguised as a warm-up exercise and a post-test disguised as an extra data-collection session were also conducted to test the student's improvement. The results showed that the average time-on-task decreases in the post-test compared to the pre-test for every subject, but since the contents of the pre-test and post-test were a subset of what the students practice with the system, it is hard to tell whether the improvement was due to short-term memory. In [**80**], similar pre-test and post-test exercises were carried out to evaluate the educational value of a vocabulary learning system. The post-test was conducted several days after the students had used the system, to minimize the effect of short-term memory. Such tests are informative, but at the same time very expensive to conduct. With the recent rise of crowd-sourcing and examples of using crowd-sourcing to evaluate conventional dialogue systems [**81**], researchers in the CALL field have started to look into this new possibility. [**82**] investigated the validity of using crowd-sourcing to conduct CALL system evaluation. The results showed that it is not very difficult to identify "scammer" workers, and most of the data they collected were from the "serious" workers.

While the power of crowd-sourcing may bring subjects and data collection to a lower cost, there still exists the problem of metrics. As we have mentioned above, questionnaires and pre-test/post-test exercise are two commonly used, and probably the only evaluation methods typically used. The contents of the questionnaires and the tests are usually put together in an ad-hoc way for each particular system, which makes it hard to compare horizontally among different systems. It would be a useful but challenging future research topic to establish a metric or a standard questionnaire, so that different CALL systems can be more easily compared with each other.

7.2.5 Natural tutoring

In addition to computer-assisted language learning, the last future direction of possible interest is to improve the tutoring mechanism embedded in the dialogue manager FAUNA towards a more naturally usable design. The current tutoring mechanism requires the developer to use special commands to intervene and correct the system behavior. If these operations can be replaced by natural commands, the tutoring would become more straightforward. Furthermore, with a natural tutoring capability, end users will be able to tutor the dialogue systems and personalize the system behavior.

The challenges lie in two aspects. First, language understanding: describing and correcting an undesired system behavior involves complex language. Understanding such language requires a generic and robust language understanding module. Second, since the system's reply is a result of a sequence of small decisions, pinpointing the exact step in which the mistake takes place is also challenging. One possible way is to create a graph where each node represents one decision. Then the graph can be searched according to the confidence score to identify the node that leads to the least confident path. This decision could then be reversed to test whether a different reply can be derived as a consequence.

7.3 Conclusion

In this thesis, several systems for interactive games to help a student learn a foreign language were developed, including reading, translation, question-answering and dialogue. We designed a framework to allow different language processing modules to cooperate and be shared among different applications, as well as the modules to perform various types of language processing, such as template-based content generation, transformation of meaning representation frames, context resolution via alignment, entity-constraint-based dialogue management, user simulation with personality, *etc.* Although the four games successfully developed using the framework and the modules focused on the language pair English-Chinese, we have shown that the modules are language-independent, and thus the games can be easily reversed to teach English to Chinese speakers. Likewise, it is also possible to swap in other L1 or L2. The techniques are also suitable for developing more interesting language learning systems. We hope this thesis can provide some inspiration for researchers and product developers to reach forward for the next generation of language learning software.

Appendix A DCTL Manual

A.1 Overview

DCTL (Dialogue Control) is a script to coordinate various language processing units. Although the name implies dialogue processing, the use has been generalized beyond dialogue.

The object of the processing indicated by the DCTL, as well as the DCTL script itself, is a GALAXY frame ("frame" by short). A Galaxy frame is enclosed by a set of curly braces, and starts with its type followed by its name. The type can be one of the following three types: "c" for clause, "p" for predicate, and "q" for topic (quantified set). The content of the frame includes two types of elements: predicates and keys. A predicate is a p-type frame led by a "*:pred*". Multiple predicates are allowed in one Galaxy frame. A key has a name and a value. The name can be any string that starts with a colon except the special key, "*:pred*", and the value can be an integer, a string, a double, another Galaxy frame, or a list of any of the above value types.

The execution of a DCTL script takes in an input frame called *input token*, and outputs an output frame called *output token*. The entire execution from receiving an input token to producing an output token is called a *turn*. All the processing that is gone through in a turn are called *operations*. A *dialogue state*, in the format of a frame, is created to store temporary information during the execution. The elements in the input token, output token, and the dialogue state are conventionally called *variables*. Predicates do not usually appear as top level elements in these three frames. The dialogue state is shared by all operations in the script, unless special operations are used. The initial dialogue state does not necessarily contain variables from the input token, nor does the dialogue state become the output token, unless the corresponding option is set. After the completion of one turn, the dialogue state is destroyed. Variables in dialogue state do not automatically transfer to the next turn, unless specified. Variables which need to be used across turns should either be stored in the *session state*, or be specified as *state variables*.

The execution of the operations in a turn follows the order of the rules in the DCTL script. Each rule represents a conditioned operation; when the condition test against the dialogue state yields *TRUE*, the operation is executed. The rules are organized into groups, called *rule lists*, represented by a Gal_List using parentheses. The execution of a turn always begin with the rule list *:rules*. Special operations allow the execution to jump to other rules lists defined in the script, and execute these essentially like a subroutine call.

The source code of loading and execution a DCTL can be found in $GALAXY_ROOT/nlcore/src/libturner. The library also includes code for common operations. Code for domain dependent operations should be found in the source code directory for each application domain under $GALAXY_ROOT/tiny_nl/src.

A.2 Running a DCTL script

As of the time this manual was written, a DCTL script can be run by two executable programs. *Tiny_brain*, which is a python wrapper, loads a DCTL script, and opens a port to listen for incoming input tokens via XMLrpc. It is used when the DCTL is a part of an entire system. *New_turnmanager* obtains input tokens from text files or standard input, and outputs into another file or standard output. It has no ability to connect to recognizers or synthesizers, but is suitable for offline process and debugging.

The two programs have different arguments. Slightly different options should be specified in the DCTL script as well. But it is possible to include options necessary for both programs in one DCTL script, so that the script can be run by either program.

Running with tiny_brain

The source code of tiny_brain is located in $GALAXY_ROOT/tiny_nl/python-tinybrain. To run, use the following command:

> python $PYTHONPATH/tiny_brain_server.py --port *port_number* --domain *domain_name* --dctl *DCTL_filename* --fork

The command creates a server and listens at the given port number. A *domain_name* is necessary, but does not have any actual effect. The argument --fork allows forking new threads for every incoming token to handle simultaneous requests. But, due to the nature of forking, the forked threads are not debuggable. This argument should be omitted if the program is run in the debug mode.

To run a DCTL script properly with tiny_brain, a list of return variables needs to be specified in the script using *:return_variables* at the top level, or to specify *:always_copy_token* in the initial frame to output everything in the dialogue state. The latter is not recommended as it would increase unnecessary communication cost.

Running with new_turnmanager

The source code of new_turnmanager is located at $GALAXY_ROOT/tiny_nl/src/new_turnmanager. To run with a text file input, use the command:

```
new_turnmanager -dialogue_script DCTL_filename
```

The commonly used optional arguments are listed below.

-dctl_file *DCTL_filename*	Same as *-dialogue_script.*
-stdin	Use standard input to provide input strings. The input string is wrapped with an empty frame with a key *:input_string* to serve as the input token.
-utts *input_filename*	Use a text file to provide inputs. The text file can be in one of the two modes. In the string mode, each line of the text file is the input string for one turn. The input token is a frame with a key *:input_string*. In the frame mode, the input file should start with "FRAMES" as the first line, followed by a frame which contains a key *:frames* with a list of input frames as value. Each input frame is the input token for one turn.
-n *n*	Combine with argument *-utts*. Use only *n* inputs from the input file.
-starti *i*	Combine with argument *-utts*. Start loading from the *i*-th input string/frame in the input file.
-copy_token	Clone the input token to be the initial dialogue state. Recommended for most situations.

-out_file *output_filename*	Specify the output file.
-out_state	Output the entire dialogue state at the end of each turn into the output file. The dialogue states are separated by a line of dashes.
-out_key *key_name*	Output only the value of one specific key at the end of each turn into the file.
-text_conversation	Print out the input string (:input_string) and reply string (:reply_string or :output_string) from the dialogue state at the end of each turn, and wait for any keyboard input to continue to the next turn.

The input token of each turn only contains the input string/frame from the text file or the standard input. To allow variables created/modified in the previous turn to be carried over to the next turn, they must be specified under the key *:history_variables* in the initial frame of the DCTL script (see next section).

A.3 DCTL script

A DCTL script file is one GALAXY frame, called the "DCTL frame". The type and name of the frame, as well as those of any subframes do not carry any actual meaning. Conventionally, different subframes may have different frame names, but it is not a part of the DCTL syntax.

The script consists of three parts: top-level options, an initial frame, and rule lists. The top-level options and the initial frame specify the initialization and global parameters. The rule lists, which are the core component of the DCTL frame, specify the conditioned operation sequences. Following is an example that parses the input string. The detailed meaning of the keys in these rules will be explained later in this section.

```
{c dctl
  :return_variables ( ":hub_session_vars" ":dispatches" )
  :initial_frame {c frame
     :always_copy_token 1
     :domains "english" }
  :rules (
        {c rule
            :conditions ":input_string"
            :variables {c variables
                :rules ":parse " }
            :operation "subroutine_call" } )
    :parse_and_paraphrase (
        {c rule
            :variables {c variables
                :input_key ":input_string"
                :output_key ":parse_frame"
                :domain "english" }
            :operation "create_frame" } ) }
```

Top-level options

Top-level options are written as top-level keys in the DCTL frame. Two options are currently available.

:insert_file ("filename1" "filename2" ...)	Include other DCTL files. The rule lists defined in other DCTL files can be called. If rule lists with the same names exist in both the current DCTL file and the inserted DCTL files, the lists in the current DCTL file are retained. Top-level options and the initial frame in the inserted DCTL files are ignored. However, the :insert_file option in the inserted DCTL files is effective.
:return_variables (":key1" ":key2" ...)	Effective option when running with *tiny_brain*. Specify the keys in the dialogue state to be sent out at the end of a turn in the output token. If this option is not specified, and *:always_copy_token* is set in the initial frame, the output token would be a copy of the dialogue state plus the session state at the end of the turn.

Initial frame

Initial frame is the frame under the key *:initial_frame*. Keys inside the initial frame are used for global settings and initialization.

Global settings

:always_copy_token *binary*	If this key is set with a nonzero value, the input token is copied to be the initial dialogue state for each turn (except reserved keys for system use, e.g. *:hub_session_vars*). If no return variables are specified, the final dialogue state is copied to become the output token. It is recommended to set this key in order to access the information from the input token in an easy way. At the same time, it is also recommended to set the return variables in order to reduce unnecessary data communication between servers.
:history_variables ("*:key1*" "*:key2*" ...)	Effective option when running with new_turnmanager. Specify keys to be carry over to the next turn.
:state_variables ("*:key1*" "*:key2*" ...)	Specify the state variables. State variables are a subset of session variables, whose values can be carried over turns. The difference between state variables and normal session variables is that a finite history of the values of the state variables is maintained as well, while normal session variables only have one current value stored. State variables are necessary to perform meta history commands like "scratch_that" (roll back one turn).
:auto_retrieve ("*:key1*" "*:key2*" ...)	Automatically retrieve the most recent value of the state variables in the beginning of a turn, and put them into the dialogue state.

:use_hub_session_tm_state *binary*	Store the state variables in a special space in the session state. Recommended to be set if the application needs to perform meta history commands.
:subturn_indicators (":*key1*" ":*key2*" ...)	Specify the sub-turn indicators. If the input token contains any of the sub-turn indicators, the turn is not considered as a new turn, *i.e.*, the history is not shifted.
:convert_import "*encoding*"	Convert the encoding of the input token to the specified one. The encoding of the raw input token is assumed to be UTF-8.
:convert_export "*encoding*"	Convert the encoding of the output token to the specified one. The encoding of the raw output token is assumed to be the same as that of the input token after conversion according to *:convert_import*.
:session_vars (":*key*")	Specify the name of key to represent the session state. *Not* recommended to set to values other than ":*hub_session_vars*".
:logdir "*directory*"	Specify the logging directory.

Initialization parameters

:domains "*domain1 domain2* ..."	Initialize the grammar domains for parsing. All grammar domains used explicitly in the DCTL script need to be specified, including any from inserted DCTL files. Grammar domains used implicitly, e.g. through DSPEC, need not be specified.

:languages "*lang1 lang2 …*"	Initialize the languages for language generation and rewrite. All languages used explicitly in the DCTL script need to be specified, including any from inserted DCTL files. Languages used implicitly, e.g. through DSPEC, need not be specified.
:dialogue_spec "*DSPEC_filename*"	Initialize the dialogue manager using the specified DSPEC file. Necessary if any operation related to dialogue management is used.
:user_sim_spec "*USIM_filename*"	Initialize the user simulator using the specified USIM file. Necessary if any operation related to user simulation is used.
:user_sim *binary*	Enable/disable user simulation initialization.
:fst_files {c fst_files :fst_dir "*directory*" :preload "*fst_file1 fst_file2 …*" }	Specify the FST directory and preload FST files.

Rule lists

Rule lists are written using Gal_List (enclosed by parentheses), in which each element is a rule frame. The name of the rule lists, *i.e.* the key of the list, can be arbitrary, except for the root list, which should be named as ":rules".

Each rule is a frame with a required key *:operation* to specify the name of the operation. The detailed syntax of the rule frame is as follows.

:operation "*op*"	Required. Specify the name of the operation.
:conditions "*logical_expr*"	Specify the conditions under which the operation is triggered to execute. The testing is conducted against the dialogue state, unless special syntax is used to indicate variables from the session state. See below for more information on the

	logical expressions.
:variables {c variables }	Include parameters for the operation. The parameters are usually operation-specific. Followings are two operation-independent parameters.
:log ":*key1* :*key2* ..."	Log the values of the specified keys at the end of the operation, if the operation is triggered.
:control ":return"	End the execution of the current rule list after the operation, if the operation is triggered. The value of the parameter is fixed.
:retrieve_session_vars ":*key1* :*key2* ..."	Copy the variables from the session state into the dialogue state. The retrieval is performed before the testing of the conditions, and thus is performed regardless of the testing result of the conditions.
:store_session_vars ":*key1* :*key2* ..."	Copy the variables from the dialogue state into the session state after the operation, if the operation is triggered.

Logical Expressions

The syntax for the logical expressions is described below. The basic expressions can be combined using conjunctive operator "&" and disjunctive operator "|". Parentheses are allowed to group the expressions for priority. If the keys have a prefix "hub_session", the value of the key is looked up in the session state, rather than the dialogue state. For binary operations, the space on the right hand side of the operator is optional.

:*key*	Existence test of the key.
!:*key*	Inexistence test of the key.
:*key str* :key =str	Equality test of the value of the key. The result is TRUE if and only if the value of the key is a string and strictly matches *str*, or the value of the key

	is a frame and the name of the frame strictly matches *str*.
:key !*str*	Inequality test of the value of the key. The result is FALSE if and only if the value of the key is a string and strictly matches *str*, or the value of the key is a frame and the name of the frame strictly matches *str*.
:key int *:key =int* *:key* !*int* *:key >int* *:key <int*	Numerical comparison of the value of the key. The result is FALSE if the value of the key is not an integer.
:key %str	Substring test of the value of the key. The result is TRUE if the value of the key is a string and contains the substring *str*.

A.4 Common DCTL operations

This section lists some frequently used operations. The source code of all the operations included here can be found at $GALAXY_ROOT/nlcore/src/libturner.

The operations can be classified into three categories: control operations, utility operations, and language processing operations. Although most parameters of the operations have default values, it is good courtesy to specify them under *:variables* in order to make the DCTL script more readable.

Control operations

Operation: subroutine_call
Variables:

:rules "*:rule_list*"	Required. The rule list to call.

Description: call the rule list named *:rule_list*. The execution of *:rule_list* does not own its private dialogue state, *i.e.*, the same dialogue state used in the execution of the current operation will be used. Self calling is possible so as to perform an iteration.

Operation: hub_rule

Variables:

:rules ":*rule_list*"	Required. The rule list to call.
:in "*in_key_list*"	Input keys. See description for details.
:out "*out_key_list*"	Output keys. See description for details.
:param *param_list*	Input parameters. See description for details.
:log_in ":*key1* :*key2* ..."	Input keys to log. The values are logged at the beginning of the call. The keys refer to the keys in the private dialogue state, rather than the caller's dialogue state.

Description: call the rule list named *:rule_list*. A private dialogue state will be created for the execution of *:rule_list*. The private dialogue state starts empty. The keys specified on the *in_key_list* and *param_list* are set in the private dialogue state with specified values. At the end of the execution of *:rule_list*, the keys specified on the output key list are transferred into the original dialogue state, and the private dialogue state is destroyed. It is important to specify appropriate input and output parameter lists in order to obtain the desired result.

The *in_key_list* is a string of tokens, separated by spaces. Each token is either a single key, or a pair enclosed by parentheses. If it is a single key, and the key exists in the caller's dialogue state, the key is copied into the private dialogue state. If it is a pair, the first element in the pair specifies the name of the key to be set in the private dialogue state, and the second element specifies the value. If the value starts with a colon, it refers to the value of the corresponding key in the caller's dialogue state; otherwise, the value is treated as a string value. Following is an example:

```
:in ":single_key (:valued_key abc) (:renamed_key :original_key)"
```

:single_key is set in the private dialogue state with the value of :single_key from the caller's dialogue state, if :single_key exists in the current dialogue state. :valued_key is set with a string value "abc". :renamed_key is set with the value of :original_key from the caller's dialogue state, if :original_key exists in the current dialogue state.

The *out_key_list* is similar to *in_key_list*, except that it sets the keys back from the private dialogue state to the caller's dialogue state.

In both *in_key_list* and *out_key_list*, any key with a prefix "hub_session" refers to the keys in the session state, instead of the dialogue state.

The *param_list* has two alternative formats.

```
:param ":key1 strvalue1 :key2 strvalue2 ..."
:param ( (":key1" objvalue1) (":key2" objvalue2) ...)
```

In the first format, the string format, an alternating string list of keys and values is specified. All the values of the keys are treated as string values. No quotes are required for *strvalue*. In the second format, the keys are able to take a Gal_Object of any type as values. If it is a string object, double quotes are necessary.

The difference between *:param* and *:in* is that *:param* is designed for constant parameters that can be specified in the DCTL rule, whereas *:in* is designed for variable parameters. However, in real use, the constant string parameters are often set by *:in*.

As one of the hub operations, theoretically, the *hub_rule* operation also supports variables *:set* and *:del*, similar to the operation *nop*. However, they are not useful in most cases.

Operation: privatize_dialogue_state

Variables:

:private_dialogue_state_name ":*name*"	Name of the private dialogue state. If the key *:name* is found in the current dialogue state and carry a frame value, the frame is used as the private dialogue state.
:copy_public_dialogue_state *binary*	If a nonzero value is set, the content of the current dialogue state is copied into the private dialogue state.
:in_keys ":*key1* :*key2* ..."	If *:copy_public_dialogue_state* is not set with a nonzero value, the keys specified in *:in_keys* are copied from the current dialogue state to the private dialogue state.

Description: create a private dialogue state and use it for the following operations. If a private dialogue state name is provided, and the name is an existing key with a frame value in the current dialogue state, the frame is used as the private dialogue state; otherwise, an empty private dialogue state is created.

Operation: restore_dialogue_state

Variables:

:save_private_dialogue_state *binary*	If a nonzero value is set, the private dialogue state is saved as a subframe in the restored dialogue state. The key of the subframe is the name of the private dialogue state specified when the operation *privatize_dialogue_state* was executed. If no name was provided, an error message is printed out. If this variable is not set, or set with a zero value, the private dialogue state is discarded.
:transfer_keys ":*key1 :key2 ...*"	Copy the keys specified from the private dialogue state to the restored dialogue state.

Description: finish using the private dialogue state and restore the original dialogue state. This operation should be used in pair with *privatize_dialogue_state*; otherwise an error message is printed out. If at the end of a turn, a private dialogue state is still in use, the restoration of the original dialogue state is performed automatically.

Utility operations

Operation: nop

Variables:

:in "*in_key_list*"	Input keys. See operation *hub_rule* for syntax.
:out "*out_key_list*"	Output keys. See operation *hub_rule* for syntax.

:set *set_list*	Set keys in the dialogue state. The syntax of *set_list* is the same as *param_list* in operation *hub_rule*.
:del ":*key1* :*key2* ..."	Delete keys from the dialogue state.

Description: the operation literally means "no operation". This operation is mostly used for setting, deleting and renaming keys in the dialogue state. Setting/deleting keys can be performed by specifying variables *:set* and *:del*. Renaming can be performed by the variables *:in* and *:out*. The following two examples have the same effect, which renames the key *:original_key* into *:renamed_key*. (Strictly speaking, it is copying instead of renaming, since both *:original_key* and *:renamed_key* exist in the dialogue state after the operation.)

```
{c rule
    :variables {c variables
        :in ":original_key"
        :out "(:renamed_key :original_key)" }
    :operation "nop" }
{c rule
    :variables {c variables
        :in "(:renamed_key :original_key)"
        :out ":renamed_key" }
    :operation "nop" }
```

Operation: increment_counter

Variables:

:counter_key ":*key*"	The key of the counter. Default to *:counter*.

Description: Increment the value of the counter key by 1.

Operation: select_nth_item

Variables:

:input_key ":*key*"	The key of the input list. Default to *:list*.
:output_key ":*key*"	The key of the output. Default to *:nth_item*.
:nth *n*	The *n*-th item to be selected.

:zero_start_index *binary*	Use zero-starting for *n*. Default to *0*.

Description: select the *n*-th item of the list specified by *:input_key*, copy it and set it in the dialogue state using the key specified by *:output_key*. The index *n* starts from 1 by default.

Language processing operations

Operation: create_frame

Variables:

:input_key ":*key*"	The key of the input. Default to *:input_string.*
:output_key ":*key*"	The key of the output parse frame. Default to *:parse_frame.*
:parse_score_key ":*key*"	The key to output the parse scores.
:domain "*domain_name*"	The grammar name for parsing. If *:tag_domain* is specified, *:domain* refers to the grammar name for 2nd pass parsing.
:domain_key ":*key*"	The key of which the value is the grammar name for parsing. Default to "*:domain*"
:tag_domain "*tag_domain_name*"	The grammar name for 1st pass parsing.
:tag_domain_key ":*key*"	The key of which the value is the tag grammar name.

Description: parse the given input. The input can be a single string, or a list of strings. If the input is a single string, the output is a frame. If the input is a list of strings, the output is a list of frames. If *:tag_domain* is specified, two-pass parsing is performed. The operation outputs two hidden variables *:parse_status* and *:parse_frame_domain*. The value of *:parse_status* can be "FULL_PARSE", "ROBUST_PARSE", or "NO_PARSE". The value of *:parse_frame_domain* is the value of *:domain* inside the parse frame. If the value of *:parse_frame_domain* is "local", it usually indicates no parse, or special inputs.

Operation: paraphrase_frame

Variables:

:input_key ":*key*"	The key of the input. Default to *:parse_frame.*
:output_key ":*key*"	The key of the output. Default to *:paraphrase_string*, if the :language_key is not ":synth_lang"; otherwise, *:synth_string*.
:domain "*domain_name*"	The domain of the paraphrase. Note, this is not the grammar name.
:domain_key ":*key*"	The key of which the value is the name of the domain.
:language "*language_name*"	The name of the language to generate into.
:language_key ":*key*"	The key of which the value is the name of the language to generate into.
:postframe ":*key*"	Keep the post-frame and output using the specified key. The post-frame is a frame with ":silent" as values for the elements that have been paraphrased to indicate that they are "consumed". It is useful for testing the coverage of the generation rules.

Description: Generate strings from the given frames. If the input is a single frame, the output is a single string. If the input is a list of frames, the output is a list of strings. The domain in the variables is not the grammar domain, but a domain defined in the GENESIS catalog. If the input object is found and the generation fails, a string "NO PARAPHRASE" is produced.

Operation: generate_kv_frame

Variables:

:input_key ":*key*"	The key of the input kv-strings. Default to *:kv_string*.
:output_key ":*key*"	The key of the output kv-frames. Default to *:kv_frame.*

Description: transform the input kv-strings into kv-frames. If the input is a single kv-string, the output is a single kv-frame. If the input is a list of kv-strings, the output is a list of kv-frames.

Operation: apply_rewrite_rules

Variables:

:input_key ":*key*"	The key of the input.
:key "*:key*"	Same as *:input_key*
:output_key ":*key*"	The key of the output. If not specified, the rewrite is performed in place.
:rewrite_catalog "*name*"	The rewrite catalog, which is the same as "domain of interest".
:domain "*domain_name*"	The grammar name to look up the associated catalog file.
:language "*language_name*"	The language.

Description: rewrite the input using the given rewrite catalog.

A.5 Advanced techniques

Control flow

In the DCTL script, branching and iteration can be realized. Branching is straightforward with the conditions as follows:

```
:rules (
   {c rule
      :conditions ":x"
      :variables {c variables
               :rules ":then" }
     :operation "subroutine_call" }
   {c rule
      :conditions "!:x"
      :variables {c variables
               :rules ":else" }
     :operation "subroutine_call" } )
:then (
      ...
      ... )
:else (
      ...
      ...)
```

Iteration is a bit more trickier. Following is an example.

```
:rules (
   {c rule
     :variables {c variables
               :set ":counter 0" }
     :operation "nop" }
   {c rule
      :variables {c variables
               :rules ":loop" }
      :operation "subroutine_call" } )
:loop (
   ...
   ...
   {c rule
      :variables {c variables
               :counter_key ":counter" }
      :operation "increment_counter" }
   {c rule
      :conditions ":counter < 10"
      :variables {c variables
               :rules ":loop" }
      :operation "subroutine_call" } )
```

Using existing rule lists

Several rule lists have been written for common functionality. They can be called directly with appropriate DCTL file inserted.

Rule list: :dialogue_turn

DCTL file: $GALAXY_ROOT/galaxy/System/dctl/dialogue_turn.dctl.frame

Functionality: perform an entire dialogue process, including language understanding and generation.

Parameters:

:enable_backdoor *binary*	Enable/disable backdoor
:enable_meta_commands *binary*	Enable/disable meta commands such as "clear history", and "scratch that".
:select_best_hyp_by_parse *binary*	If a nonzero value is set and the input is an N-best list, the best input string is selected according to the parsing result, *i.e.*, the top hypothesis with a full parse is selected.
:score_planner_reply_rules *:rule_list*	Specify the rule list to score the resulting dialogue replies of the N-best input. The specified rule list should output a key *:planner_reply_score* of which the value is a list of scores. If no rule list is specified, *:select_best_hyp_by_parse* is forced to be 1.

Input keys: depending on the dialogue specification, other inputs are possible. The following two are most common.

:input_string	A single input string.
:nbest_list	A list of N-best input strings.

Output keys: depending on the dialogue specification, other outputs are possible. The following two are most commonly available.

:reply_string	The system reply.
:selected_input_string	The selected input string, if the input is an N-best list.

Rule list: :gui_output

DCTL file: $GALAXY_ROOT/galaxy/System/dctl/gui_output.dctl.frame

Functionality: send GUI dispatches for HTML update and the conversation history box update.

Input keys:

:html_string	The HTML update.
:input_string	The value of this key will be displayed in the conversation history box followed by "User:"
:reply_string	The value of this key will be displayed in the conversation history box followed by "System:"

Rule list: :synthesis

DCTL file: $GALAXY_ROOT/galaxy/System/dctl/synthesizers.dctl.frame

Functionality: send a synthesis dispatch.

Parameters:

:synthesizer "*name*"	The name of the synthesizer. Available choices are: CAS (Chinese GB), Dectalk (English), ITRI (Chinese BIG5)

Input keys:

:synth_string	The string to synthesize.

Adding new DCTL operations

Adding new DCTL operations should follow the steps below:

1. Include “turn_manager.h” in the source file.

2. Define the operation translator.

```
struct DC_OPERATION new_translator[] = {
   {“operation_name”, function_name},
   {NULL, NULL}
 };
```

3. Implement the function. The return value is *DIALOGUE_CONTINUE* for most cases. Please be advised that although the dialogue state is fully accessible without specifying any variables in the variables frame, it is good practice to not access/produce keys other than those specified in the variables frame.

```
DC_RESULT function_name(DIALOGUE_MANAGER* dc, Gal_Frame variables) {
   return DIALOGUE_CONTINUE;
}
```

4. In $GALAXY_ROOT/nlcore/include/tina/dialogue.h, add the new operation translator.

```
extern struct DC_OPERATION new_translator[];
struct DC_OPERATION *AllExternalFunctionTranslators[] = {
   ....,
   new_translator
};
```

5. In the makefile of the executable programs (tiny_brain and new_turnmanager), add the library with the new code.

6. Make the libraries and executables.

```
cd $GALAXY_ROOT/nlcore
make install
cd $GALAXY_ROOT/tiny_nl/src/new_turnmanager
make install
cd $GALAXY_ROOT/tiny_nl/python_tinybrain
make [debug]
```

Appendix B Dialogue Specification (DSPEC)

B.1 Overview

The dialogue specification (DSPEC) is the domain specification file for the dialogue manager FAUNA. The specification includes the declaration of the knowledge sources, the nations, the entity structures, etc.

To use FAUNA properly, a DSPEC is necessary, as well as an appropriate DCTL script. Currently, FAUNA does not have a stand-alone executable program. It has to be run in the DCTL framework. The DCTL rule list for a typical dialogue turn, including language understanding and language generation is specified in file

```
$GALAXY_ROOT/galaxy/System/dctl/dialogue_turn.dctl.frame
```

In the top DCTL script of a typical dialogue system, this file can be specified as an insertion file. Please note that the DSPEC should be specified in the initial frame, also defined in the DCTL script.

Running a DCTL script with FAUNA produces a separate log file for FAUNA. The directory of the log file is specified according to the DSPEC. If statistical inferences are utilized in the dialogue domain, a separate statistical inference specification is needed, as well as a specified location for reading/writing data.

The entire DSPEC is written in one file, in the format of a Galaxy frame. The content of the frame can be divided into five sections: the top-level options, knowledge source declarations, nation declarations, entity declarations, and meta information specifications. Following is an illustrative DSPEC.

```
{c DSPEC
  :init_domain "domain"
  :goals {q goal_entity
       :domain "domain" }
  :knowledge_sources (
            {q user
               ... }
             {q database
               ... })
        :nations (
             {q english
                ... })
        :entities (
             {q goal_entity
                ... }
             {q another_entity
                ... })
        :meta_information {c meta
            ... }}
```

B.2 Top-level options

Following is a list of available top-level options.

:init_domain "*domain*"	The initial domain to start with. Required even for single domain dialogues.
:stat_inference_spec "*filename*"	The specification for the statistical classifiers.
:stat_inference_data "*file_prefix*"	The file prefix (including path) for loading and writing classification data. Each classifier would create a separate file for data using the prefix.
:no_learning *binary*	No new data points are written into the data file if this option is set with a nonzero value.
:log_with_dctl *binary*	Create the log file in the same directory as the DCTL logs, if this option is set with a nonzero value.
:bkdoor_log_dir "*directory*"	Specify the directory for log files. A separate log file is created for each session. This option should not appear together with *:log_with_dctl* with a nonzero

	value.
:keep_communication_eforms *binary*	Save the input and output eforms in FAUNA's state, if a nonzero value is set. Default to 0. Be advised that these eforms can be huge and costly to transmit between servers.
:knowledge_source_priority ("*ks1*" "*ks2*" ...)	Specify the priority of the knowledge sources, in case that two knowledge sources can provide the same type of knowledge. Knowledge sources with higher priority should be listed earlier. "User" should always be the lowest knowledge source.
:goals {q *goal_entity* :domain "*domain_name*" } :goals ({q *goal_entity1* :domain "*domain1*" } {q *goal_entity2* :domain "*domain2*" })	Required. Specify the goal entities for each domain. The value can be a single frame in a single domain dialogue, or a list of frames in a multi-domain dialogue (not implemented yet).

B.3 Knowledge sources

Knowledge sources are the places where knowledge can be obtained, for example databases, local libraries, and the user. The declaration of the knowledge source includes its nation, which indicates its input/output language processing, its source, and the set of knowledge it possesses.

Declaring knowledge sources

To declare the knowledge sources, use the key *:knowledge_sources*. Its value can be a single frame, or a list of frames. Each frame has the name of the knowledge source as its name, shown as follows.

```
:knowledge_sources {q ks_name
      ... }
:knowledge_sources (
      {q ks_name1
```

```
        ... }
    {q ks_name2
        ... })
```

The content of the frame includes the following keys.

:nation "*nation_name*"	Specify the nation of the knowledge source.
:source {q *source* }	Specify the source of the knowledge source. Three types are available: *local* for local libraries for which the handlers can be called directly; *external* for external programs that need dispatches (not implemented); *gui* for "user". The content of the frame can be used to specify any necessary parameters.
:knowledge {q *knowledge_name* ... } :knowledge ({q *knowledge1* ... } {q *knowledge2* ... })	Declare the pieces of knowledge the knowledge source can provide with. The value can be one frame or a list of frames depending on the number of types of knowledge. The knowledge source "user" does not need to declare the knowledge.
:request_prompt ("*task1*" "*task2*" ...)	Specify the list of tasks for which, when finished, a prompt would be sent to the knowledge source.
:max_option_length *int*	Specify the maximum length of the pending list to produce the continuant *select_n* with the reply message *need_select*. If the pending list is longer than this number, the continuant *too_many_options* is generated.

Declaring knowledge

Each piece of knowledge can correspond to an entity, which has the same name as one of the entities declared in the specification, or to an attribute, which has the same name as one of the knowledge domains of the attributes. The declarations of both types are similar, except for one key.

The declaration of each piece of knowledge is one frame. A knowledge source might contain a single knowledge frame, or a list of knowledge frames. The name of a knowledge frame is the name of the piece of knowledge.

```
{q knowledge_name
    :prerequisite ( ... )
    :initializer {c eform
        ... }
    :handler "..." }
```

Following is a list of possible keys in the knowledge frame.

:attributes ("*attr1*" "*attr2*" ...)	If the piece of knowledge corresponds to an entity, list the attributes of the entity that the knowledge source can provide. Usually, if the knowledge source is a database, the attributes correspond to the fields of the table. If the piece of knowledge correspond to an attribute, ignore this key.
:prerequisite (("*attr11*" "*attr12*" ...) ("*attr21*" "*attr22*" ...) ...)	Same as the ":primary_keys" in older versions. Specify the prerequisite to perform a handler call. The value of the key is a list of lists. Each inner list represents a conjunctive condition. The inner lists are combined disjunctively. The equivalent logical form of the value is "(*attr11* & *attr12* & ...) | (*attr21* & *attr22* & ...) | ..."
:initializer "*name*" :initializer {c eform :function "*name*" :arg {c param ..... } }	Specify the initializer of the piece of knowledge. Specify the name of the initializer in a string format if no parameter is necessary. Otherwise, use a frame to specify both the name and the parameters. Currently only applicable for local knowledge sources.

:handler "*name*"
:handler {c eform
    :function "*name*"
    :arg {c param
      ..... } }
	Specify the handler for inquiring/resolving the piece of knowledge. Specify the name of the handler in a string format if no parameter is necessary. Otherwise, use a frame to specify both the name and the parameters.

:comparator "*name*"
:comparator {c eform
    :function "*name*"
    :arg {c param
      ..... } }
	Specify the handler for comparing two instance of this piece of knowledge. Specify the name of the comparator in a string format if no parameter is necessary. Otherwise, use a frame to specify both the name and the parameters. Currently only applicable for local knowledge sources.

:summarizer "*name*"
:summarizer {c eform
    :function "*name*"
    :arg {c param
      ..... } }
	Specify the handler for summarizing a list of instances of this piece of knowledge. Specify the name of the summarizer in a string format if no parameter is necessary. Otherwise, use a frame to specify both the name and the parameters. Currently only applicable for local knowledge sources.

Implementing knowledge source handlers

For the local knowledge sources, the function prototype of the handlers, including initializers, comparators and summarizers is given below.

```
typedef Gal_Frame (ks_handler_func) (Gal_Frame query_frame, Gal_Frame
session_info);
```

The first argument *query_frame* contains the content for querying, resolving, comparing and summarizing. Any parameters specified in the DSPEC are contained in *query_frame* as well. Depending on the type of the handler, the content of *query_frame* is different, which will be explained below.

The second argument *session_info* contains any session information that is stored for the entire session. Each knowledge source owns a separate *session_info* frame. Information written into *session_info* can be accessed in the next call of any type of handler declared for this knowledge source.

The return value of the handler is a frame or NULL. In the case it is a frame, it should be a new frame, rather than the *query_frame*. The detailed content of the *query_frame* and return frame is explained below.

Initializer: *query_frame* only contains the parameters specified in the DSPEC. If the initialization fails, a nonzero integer value should be set in the return frame using key KS_REPLY_FAIL_KEY. Otherwise, the return frame can be NULL.

Handler: if the knowledge source belongs to a nation other than "eform", the handler receivers information after language generation, and sends back information for language understanding. In other words, the keys in *query_frame* correspond to the input perceptions of the nation, and the contents in the return frame should be able to be passed to the specified language understanding procedure.

If the knowledge source belongs to the nation "eform", the query_frame and return frame look like the following:

Query_frame :

```
{q departure_date_query
  :departure_date {... }
  :parameters ... }
```

Return frame:

```
{q departure_date_reply
  :departure_date {....} }
```

The return frame can optionally contain the following specially reserved keys:

KS_REPLY_TO_USER_KEY {c ... }	A reply message to the user.
KS_REPLY_WAIT_FOR_USER_KEY binary	Break the task loop and wait for the user's response if this key is set with a nonzero integer. Usually appears together with KS_REPLY_TO_USER_KEY.

KS_REPLY_MISDIRECTED_KEY {c ... }	A frame containing key-value pairs from the *query_frame* which are possibly misdirected.

Comparator: *query_frame* contains two keys in addition to the parameters specified in the DSPEC: *:ref* and *:compare*. The value of the key *:ref* is the reference object. The value of the key *:compare* is the object to be compared with the reference. The return frame should contain the key KS_RESULT_KEY with a positive integer value if the comparing object is greater than the reference object, with a negative integer value if the comparing object is less than the reference object, with a zero value if they are equal.

Summarizer: *query_frame* contains a key *:summarize* with a list of objects as values, as well as a key *:ref*, of which the value can be used as a reference when doing summarization. The return frame should contain the key KS_RESULT_KEY with a frame value. The frame is the summarization result in whatever format the developer would like. (The summarization result usually appears in the system reply, and thus should be consistent with the subsequent language generation process.) The frame can also include the following optional keys.

:pending_update (...)	Specify a new pending list to replace the current one in FAUNA. This update is necessary when the summarization changes the order of the pending objects, so that a mismatch might result between what is replied to the user and what is stored in FAUNA.
:nrecommended *int*	The number of most recommended options.
:too_many *int*	If this option is set with a nonzero integer, the continuant *too_many_options* would replace the continuant *select_n*.

B.4 Nations

Nations indicate the language processing parameters for the knowledge sources. The nomenclature comes from the nations in the real world: people from different nations speak different languages.

The key *:nations* is used to declare the nations, followed by a single frame or a list of frames. For each frame, the name of the frame is the name of the nation. It includes two keys, :input_perceptions and :output_perceptions to specify the input and output perceptions (modalities). The rest of the keys in the frame should correspond to the perceptions. Following is an example:

```
:nations {q human
    :input_perceptions ( ":hears" ":reads" )
    :output_perceptions ":speaks"
    :hears {q string :genesis_language "English" }
    :reads {q string :genesis_language "html" }
    :speaks {q string
            :tina_grammar "English"
            :kv_lang "kv" } }
```

The keys :input_perceptions and :output_perceptions can carry a single string value or a list of strings. The words "input" and "output" are respective to the knowledge source, rather than FAUNA. The keys corresponding to the perceptions carry frame values. In the situation of multi-domain dialogues (not implemented), they can have a list of frames as values, with each frame specifying different parameters for different domains.

The frame for each perception has the format of the perception as its frame name. The possible choices are *string* and *eform*. A handler can be specified to handle the understanding/generation of the perception. The handler function has the following prototype, which takes in a frame containing raw input/output, and returns a frame containing the processed input/output.

```
typedef Gal_Frame (nation_handler_fun) (Gal_Frame frame);
```

Alternatively, an input perception can use GENESIS as the language generation module, and an output perception can use TINA as the language understanding module.

A perception can be declared as a parasite modality by using the key *:parasite_modality*. Perceptions that are not parasite modalities are called main modalities. Outputs from the parasite modalities alone do not trigger the dialogue management process; they have to rely on other outputs from main modalities to trigger the dialogue management. Parasite modalities and the main modalities can be asynchronous. Timeout parameters should be set for meaningful usage.

Below lists the possible keys for the perception frames.

:handler "*handler_name*"	The handler to call to process the input/output.
:domain "*domain_name*"	Specify the domain to apply the parameters in the frame. The "domain" refers to application domains, rather than grammar domains. Not necessary to specify the domain if it is a single domain dialogue.
:genesis_language "*language*"	The language to use for generation using GENESIS. The "domain" that GENESIS requires comes from the current application domain of the dialogue.
:tina_grammar "*grammar_name*"	The grammar for parsing using TINA. If *:tina_grammar_tag* is present, the grammar refers to the 2nd-pass grammar.
:tina_grammar_tag "*grammar_name*"	The tag grammar for 1st-pass parsing using TINA.
:kv_lang "*language*"	The GENESIS language to produce a key-value representation from a parse frame.
:kv_in *binary*	If set with a nonzero value, the input string is assumed to be in a kv-string format. Parsing and kv paraphrasing is skipped.
:parasite_modality *binary*	Declare the perception to be a parasite modality.
:asynchronous_lifetime *int*	The maximum time in seconds to allow the output from this modality to wait for an upcoming output from the main modality.
:asynchronous_latency *int*	The maximum time in seconds to allow the output from this modality to be behind the output from the main modality.

B.5 Entities

The entity section declares the entity types and their relationships in the domain. The declaration starts with the key *:entities*, followed by a single entity frame, or a list of entity frames.

```
:entities {q entity_name
  ... }
```

Or,

```
:entities (
    {q entity1
      ... }
    {q entity2
      ...} )
```

The name of the entity frame is the name of the entity type. The content of the entity frame consists of several parts: definition, completion condition, governing relationship, modifiers, customized actions, and commands. These parts are explained in detail below.

Definition :definition

The definition of the entity type is declared using the key :definition with a frame as the value. The name of the frame does not carry any meaning. Each key in the definition frame is a member of the entity type. The string value of the key indicates the type of the member. The type can be one of the simple types: "int", "float" and "string", another entity type, or a list of the above types using the prefix "list-of-". Initial values can be specified for members with simple types using a colon. An example is given below. The member *:count* is declared as an integer type with an initial value 0. The type of the member *:users* is a list of another entity type "user".

```
:definition {c definition
  :count "int:0"
  :color "string"
  :users "list-of-user" }
```

A special type of members called *attribute* is declared using the predicates. They are members that have complex information, but not complex enough to become an entity. Knowledge domains can be assigned to the attributes to allow FAUNA to contact the knowledge sources for extra processing.

```
:definition {c definition
    :pred {p destination
        :knowledge_domain "airport" } }
```

Besides the :knowledge_domain, a number of other keys are available to declare an attribute.

:alias (*"alias1" "alias2"* ...)	Specify other names representing this attribute in the incoming eforms to FAUNA. If the name starts with a colon, it is assumed to be a key; otherwise it is assumed to be a predicate.
:adopts (*"name1" "name2"* ...)	Specify the elements in the incoming eforms to FAUNA which the attribute should adopt as its child. If the name starts with a colon, it is assumed to be a key; otherwise it is assumed to be a predicate.
:knowledge_domain *"domain"*	Specify the knowledge domain of the attribute. At least one knowledge source should be able to provide the knowledge.
:format *"logical_expr"*	Specify the expected format of the content of the attribute. If the format is not satisfied, FAUNA contacts the appropriate knowledge source for a resolution. The logical expression uses the same syntax as the conditions in DCTL scripts. See Appendix A.3 for details.
:primary *binary*	Indicate whether the attribute is primary or not. A primary attribute is one that will not be removed when the user indicates "remove all constraints", or "any would be fine".
:user_ignorant *binary*	Indicate whether the attribute can be provided by the user by any chance. If this key is set with a nonzero integer value, FAUNA will never ask the user for this attribute, but instead reply with "no_match".
:auto_update *binary*	Indicate whether the attribute is automatically updated. If this key is set with a nonzero integer value, the corresponding *update_attribute* task should be implemented.

Completion condition :goal

The completion condition specifies the conditions under which the entity is considered to be complete. The condition is expressed as a logical expression using the key *:goal*. This condition is important, as FAUNA issues tasks in order to fulfill the completion condition.

The syntax of the logical expression is similar to that used in *:conditions* for the DCTL rules, with the following several extensions.

- The equality, inequality, and numerical comparison tests can be applied to values of two keys. For example, the following expression tests the value of :key1 is not equal to the value of :key2.

```
":key1 !:key2"
```

- Keys from subframes can be referred using square brackets. For example, the following expression tests the value of *:key1* under *:parent* is "abc". The brackets can be nested.

```
":key1[:parent] abc"
```

- "#:key" evaluates to the length of the value of the key. If the value is not a list, it always evaluates to 1. If the key does not exist, it evaluates to 0.
- "#keys" evaluates to the number of keys in the frame that is tested against. Keys starting with ":*" are considered as temporary keys and are not counted.

Governing relationship :governing_relationship, :inverse_governing_relationship

The governing relationships describe the discourse relationship among the members of the entity. Members in the entity are maintained across turns by default. However, if member A governs member B, any modification to member A would result in a removal of member B; *i.e.*, member B is no longer valid after member A being modified. Member A is called the governor, and member B is called the governee.

The governing relationships can be specified in two directions using the keys :governing_relationship and :inverse_governing_relationship respectively, or a combination of both. The usage of the two keys is given below.

```
:governing_relationship {c relationship
    :governer1 ( "governee11" "governee12" ... )
    :governer2 ( "governee21" "governee22" ... )
     ... }
:inverse_governing_relationship {c relationship
    :governee1 ( "governer11" "governer12" ... )
    :governee2 ( "governer21" "governer22" ... )
    ... }
```

The governors and the governees can be both attributes and normal members. Attributes are specified without a preceding colon (as they are predicates in the frames). Normal members are specified using a preceding colon (double colon if they are in the position of the keys).

The governing relationships can be conditioned in a limited way. The governors and governees can be conditioned by their values. The following example shows the syntax of attribute A governs attribute B when A's value is a and B'value is not b.

```
:governing_relationship {c relationship
    :A=a ( "B!=b" ) }
```

The syntax only supports equality and inequality of string values. If the member has a frame value (including attributes), the string test can be evaluated using the frame names. Multiple values are allowed to appear on the right hand side of the operator, separated by commas. Note that there should be no space in the whole expression. The expression

```
A=a,b,c
```

Is equivalent to

```
A=a | A=b | A=c
```

And the expression

```
A!=a,b,c
```

Is equivalent to

```
A!=a & A!=b & A!=c
```

Modifiers :modifiers

Modifiers are constraints on the attributes which have effect when there are a set of candidate entities. The only implemented modifier at the time of this manual is the superlatives.

The superlatives are specified as a list of tuples. The first element is the superlative word used to refer to the top object when a list of objects are sorted in an ascending order. The second element is the superlative word used to refer to the top object when a list of objects are sorted in a descending order. The third element is the name of the attribute that the sorting is performed on. An example is given below.

```
:modifiers {c modifiers
    :superlatives ( ( "earliest" "latest" "departure_time" )
                    ( "cheapest" "most expensive" "price" ) ) }
```

Customized actions :customized_actions

To use customized actions for the reserved tasks, task-action association must be specified using the key *:customized_actions*. For customized tasks, FAUNA looks for the action with the same name as the task by default. If the action does not share the same name as the task, it should be specified as well.

Following is a list of reserved tasks for which the actions are commonly customized.

Complete_entity, show_entity,
Add_entity, alter_entity, remove_entity,
Fill_attribute, change_attribute, drop_attribute

The task names of these reserved tasks are comprised of a verb and an object. The two tasks in the first row do not carry a parameter. To specify a customized action, any of the three ways in the following example would work, where the italic "entityname" stands for the actual name of this particular entity type.

```
:customized_actions {c actions
    :on_complete "action_name"
    :on_complete_entity "action_name"
    :on_complete_entityname "action_name" }
```

For the rest of the reserved tasks, since the tasks carry a parameter, *i.e.*, the member entity/attribute to be added, modified, or removed, the customization can be done in two levels:

specifying a customized action regardless of the parameter, or specifying a customized action for the task when the parameter is a particular member. In the first case, the specification should use the whole task name, as shown below.

```
:customized_actions {c actions
        :on_fill_attribute "action_name" }
```

In the second case, the specification should use the combination of the verb part of the task name and the name of the member. The example is given below.

```
:customized_actions {c actions
        :on_fill_destination "action_name" }
```

When implementing the customized actions, each action maps to an optional prepare function and a required action function. If the prepare function does not need customization, map it to the default prepare function "default_prepare". The prototypes and return values of the two functions are given below.

```
typedef PrepareResult (task_prepare_func) (DIALOGUE_PLANNER* planner, Gal_Frame task);
typedef ActionResult (task_action_func) (DIALOGUE_PLANNER* planner, Gal_Frame task);
```

PrepareResult	NotReady	Not ready to execute the task
	ActionReady	Ready to execute the task
	TaskAbort	The task will be removed
	NoNeedToProceed	The task will be removed
ActionResult	NotCompleted	The task has not been completed
	AttributeAdd	An attribute has been added into the entity
	AttributeDel	An attribute has been deleted from the entity
	AttributeMod	An attribute has been modified in the entity
	EntityAdd	A member which has an entity type has been added into the entity
	EntityDel	A member which has an entity type has been removed from the entity
	EntityMod	A member which has an entity type has been modified in the entity
	ModifierMod	The entity's modifier has been changed
	CompletedNoChange	The task is completed and no change has occurred to the entity

In the prepare function and the action function, the default prepare/action functions can be called as follows.

```
PrepareResult pr = default_prepare(planner, task);
ActionResult ar = default_action(planner, task);
```

Commands

The commands specify the mapping from command words to tasks. A command can be mapped to a task using a frame or a sequence of tasks using a list of frames. In each task frame, the name of the frame is the name of the task. The key :entity is required to specify the object entity of the task. If the task also takes a parameter, the key *:param* is also necessary. Other keys may be allowed if it is a customized task or has a customized action, as they will be accessible in the action function of the task. A special value "*SELF*" can be used to refer to the entity itself on which the command is issued. An example of a command "book" for the entity "flight" is given below.

```
:commands {c commands
    :book {q add_entity
            :entity "itinerary"
             :param "*SELF*" } }
```

B.6 Meta information

The section of meta information is used to specify the functional keys and values appearing in the incoming eforms to FAUNA, such as nth, truth values (yes and no), and quantifiers. Most of them have default values, but can be modified to adapt to the specific dialogue domain.

The specification of meta information uses the key *:meta_information* with a frame value. Following is a list of keys and the format of their values.

:nth_key ":key1 :key2 ..."	Specify the key(s) for "nth" in the incoming eforms for FAUNA.
:truth_value {c eform :key ":key1 :key2 ..."	Specify the key(s) for truth values in the incoming eforms in FAUNA, as well as the values for positive

:yes "value1 value2 ..." :no "value1 value2 ..." }	"yes" and negative "no".
:quant {q quant :key ":key1 :key2 ..." :def "value1 value2 ..." :dem "value1 value2 ..." :indef "value1 value2 ..." :all "value1 value2 ..." :other "value1 value2 ..." }	Specify the key(s) for quantifiers in the incoming eforms to FAUNA, as well as the values for definitive ("the"), demonstrative ("this", "that", etc.), indefinite ("a"), all ("all"), and other ("other", "else", etc.).
:modifier {q modifier :superlative ":key1 :key2 ..." }	Specify the key(s) for modifiers in the incoming eforms to FAUNA.

B.7 Specification for statistical inference

The features for the statistical classifiers are specified in a separate file for potential sharing between different dialogue domains.

The specification is written using a frame. Except for two keys *:manual_weight* and *:alpha* for specifying general coefficients, each key in the frame specifies the parameters and features of a classifier. Following is an example specification frame.

```
{c inference
    :manual_weight 2
    :alpha "0.9"
    :task_needs_confirm {c feature
        :values ( 0 1 )
        :ninput_frames 2
        :features ( ":confirmed[0]" "$core[0]" ":entity_shown[1]" ) } }
```

The key *:manual_weight* specifies the weight for the data points which are obtained from a manual correction. Usual data points have a weight of 1. The key *:alpha* specifies the fading coefficient, which takes a value between 0 and 1. The smaller this coefficient is, the less weight older data points have.

The specification for each classifier uses the name of the classifier as the key. Inside the specification are three keys. The key *:values* specifies all the possible output labels of the classifier. The key *:ninput_frames* specifies the number of input frames when calling the classification function. The key *:features* specifies the features used in the classification. Each feature consists of a name and a index which indicates from which input frame the feature value is computed. The name of the feature can be any string (not restricted to keys), and functions for computing the feature value can be defined. If no function is defined, and the name of the feature is a key, the feature value is obtained by finding the value of that key in the input frame. A special feature name "$core" is also defined, of which the value is the name of the input frame. The index of the input frame is written in square brackets.

To define functions for computing the features, each feature name should map to two functions: one for computing the feature value from an input frame, the other for computing the similarity between two feature values. The two function prototypes are given below. Feature_func returns the feature value as a Gal_Object. Feature_sim_func returns a real number between 0 and 1, in which 1 stands for identical values.

```
typedef Gal_Object (feature_func) (DIALOGUE_PLANNER* planner, Gal_Frame input_frame);
typedef double (feature_sim_func) (Gal_Object f1, Gal_Object f2);
```

To use the classifiers in the code, call the following function with the corresponding classifier name and input frames.

```
Gal_Object stat_inference(DIALOGUE_PLANNER* planner, const char* inference_name, ...);
```

An example is given below.

```
Gal_Object result = stat_inference(planner, ":task_needs_confirm", input_frame1, input_frame2);
```

B.8 Adding a domain-dependent dialogue library

A domain-dependent dialogue library includes the implementation of customized task actions, knowledge source handlers, nation handlers, and feature functions. To make the library linked with the entire environment smoothly, creating a .c file and a .h file is recommended.

The following steps describe one way to build the domain-dependent dialogue library. In this example, the directory should be put under $GALAXY_ROOT/tiny_nl/src. A more concrete example can be found at $GALAXY_ROOT/tiny_nl/src/libnewmercury.

1. Create a header file yourdomain.h in $GALAXY_ROOT/tiny_nl/include/tina. Include "tina/dialogue_planner.h".

2. Create a c file yourdomain.c under $GALAXY_ROOT/tiny_nl/src/yourdomain. Include "tina/yourdomain.h"

3. Define the task action mapping, knowledge source handler mapping, nation handler mapping, and feature function mapping.

```
static task_prepare_func t_prepare;
static task_action_func t_action;
static ks_handler_func ks_func;
static nation_handler_func nation_func,
static feature_func f_func,
static feature_sim_func f_sim_func;
TASK_OPERATION YourDomainOperation[] = {
   {"your_task", t_prepare, t_action},
   {NULL, NULL, NULL}
};
KS_HANDLER YourDomainKSHandler[] = {
   {"your_ks_handler", ks_func},
   {NULL, NULL}
};
NATION_HANDLER YourDomainNationHandler[] = {
   {"your_nation_handler", nation_func},
   {NULL, NULL}
};
FEATURE_FUNCTION YourDomainFeatureFunction[] = {
   {"your_feature", f_func, f_sim_func},
   {NULL, NULL, NULL}
};
```

4. Create arrays for the above mappings. Mappings can be defined in different files. These arrays should contain all the mappings it is necessary to include in this domain.

```
const TASK_OPERATION* YourDomainOperationAll[] = {YourDomainOperation, NULL};
const KS_HANDLER* YourDomainKSHandlerAll[] = {YourDomainKSHandler, NULL};
const NATION_HANDLER* YourDomainNationHandlerAll[] = {YourDomainNationHandler,
NULL};
const FEATURE_FUNCTION* YourDomainFeatureFunctionAll[] =
{YourDomainFeatureFunction, NULL};
```

5. Create the domain recipe.

```
DIALOGUE_DOMAIN_RECIPE YourDomainRecipe = {
  YourDomainOperationAll,
  YourDomainKSHandlerAll,
  YourDomainNationHandlerAll,
  YourDomainFeatureFunctionAll
};
```

6. In the header file, add the domain recipe.

```
extern DIALOGUE_DOMAIN_RECIPE YourDomainRecipe;
```

7. Implement all the functions.

8. In $GALAXY_ROOT/tiny_nl/include/tina/dialogue_recipe.h, add the domain recipe. Note that the domain name in quotes should be the same as the domain specified in the DSPEC (:init_domain), so that the correct recipe will be chosen to load during FAUNA's initialization.

```
#include <tina/yourdomain.h>
Extern DIALOGUE_DOMAIN_RECIPE YourDomainRecipe;
DIALOGUE_RECIPE_MAP DialogueRecipeMaps[] = {
   ...
   {"YourDomain", &YourDomainDialogueRecipe},
   {NULL, NULL}
};
```

9. In the Makefile.am, make sure the dialogue library *ldialogue* is linked. It should also contain the following lines:

```
tinaincludedir = $(includedie)/tina
tinainclude_HEADERS = \
    $(top_srcdir)/include/tina/yourdomain.h \
    $(top_srcdir)/include/tina/dialogue_recipe.h
```

10. In $GALAXY_ROOT/tiny_nl/configure.ac, add your directory to generate the Makefile.

```
AC_CONFIG_FILES ( [...
    Src/yourdomain/Makefile
   ... ])
```

11. In $GALAXY_ROOT/tiny_nl, do

```
autoreconf -fvi
make
make install
```

12. Now your library should be made. In the makefiles of the executable programs (tiny_brain and new_turnmanager), add your library, and make them.

Appendix C User simulation specification

C.1 Overview

The user simulation specification is used for instructing the user simulator to generate the user intention and the user response/suggestions. The simulator is designed to cooperate with the dialogue manager FAUNA. The input to the simulator is usually the reply eform from FAUNA, and the output of the simulator is another eform, which can then be paraphrased into a natural sentence using GENESIS, or sent back to FAUNA directly.

The simulator should be used under the DCTL framework, either via the operation "user_simulation", or via other operations which call the simulation/suggestion functions in the code. To enable simulation, in the DCTL script, the user simulation specification should be specified in the initial frame using the key *:user_sim_spec*, and the key *:user_sim* should be set with a nonzero integer value.

The specification is written in a GALAXY frame format, and is divided into two parts: specifying the scenario (user intention), and specifying the response strategies. Both are written using the notion of "templates", *i.e.*, all the possible values are specified in a compact form, and at runtime, a specific scenario or response is instantiated randomly or according to some probability distribution.

The user simulator creates a separate log file for every session. The log directory can be specified using the key *:log_dir* at the top level of the specification frame. An alternative way is to log in the same directory as the DCTL logs by using the key *:log_with_dctl* with a nonzero integer value.

C.2 Scenario

The scenario represents the user's intention. The instantiated scenario is a frame with key-value pairs representing the user's knowledge and preferences, which provide the information for instantiating user responses. Since the user simulator is usually used in conjunction with FAUNA, the keys in the scenario should be consistent with the declarations in the dialogue specification (DSPEC).

If *new_turnmanager* is used as the executable program, the scenario is instantiated automatically before receiving the first input. However, if *tiny_brain* is the executable program, the scenario should be instantiated by calling the following function in the appropriate (domain-dependent) operations. The function takes the filename of the DCTL log file for this session, and returns the initial memory state containing the instantiated scenario.

```
Gal_Frame start_user_simulator(const char*dctl_logfile);
```

The description of the scenario involves the following five top-level keys in the specification: *:scenario*, *:scenario_constraint*, *:scenario_phases*, *:scenario_post_processing*, and *:registered_ks_handler*. The usages of the five keys are explained in detail below.

:scenario

The scenario templates are specified under the key *:scenario*. The key takes a single template frame or a list of frames as its value. If a list of template frames is given, a random one is chosen to instantiate at runtime.

Inside each template frame are the keys meaningful in the specific dialogue domain. If the value of the key is a list, it indicates all the possible choices of the value. Otherwise, the key is considered to have a fixed value. The list value can have one of the following two forms:

```
:key ( value1 value2 ... )
:key ( "param=x" ( value1 value2 ... ) )
```

Both forms specify the possible values of the key. The second form provides additional parameters for instantiation. When instantiating a key with a list value, the simulator assigns a random probability to each of the possible values and produces a preference list for the key. The additional parameters are used to adjust the smoothness of the probability distribution, and/or retain only the top-N choices.

To adjust the smoothness of the probability distribution, use the parameter

```
smooth=x
```

where *x* is a positive real number. If x < 1, the effect is to smooth the probability distribution. The smaller x is, the closer the distribution is to the uniform distribution. If x > 1, the effect would be the opposite of smoothing, *i.e.*, the probabilities of the values would be more extreme.

To only keep n values with highest probabilities, use the parameter

```
nselect=x
```

where x is an integer. If x equals to 1, the instantiation result of the key becomes a single value instead of a preference list. Note that there is no space on either side of the equal sign.

If, in the final preference list, one value has a probability greater than the "single value threshold" (0.8), all the other values are discarded, and the key would have a single value instead of a preference list. Otherwise, a special value "*all*" is added into the preference list to indicate that the user does not care about the value of this key.

Following is an example of the template frames. Note that the instantiation is recursive, *i.e.*, elements in the subframes will also be instantiated. If the possible choices of the values of a key are integers within a range, it is allowed to simply specify the starting and ending integers.

```
:scenario ( {c scenario
      :source ( "nselect=1" ( "Boston" "Chicago" "New York" ) )
      :destination ( "nselect=1" ( "Boston" "Chicago" "New York" ) )
      :departure_date {c eform
          :ndays ( "smooth=0.5" ( 3 30 ) )
          :units "days later" } } )
```

:scenario_constraint

This key specifies the constraints that the scenario should satisfy. It takes a string value, which is a logical expression. The syntax of the logical expression is the same as that used for specifying the completion conditions of the entities in the dialogue specification (see Appendix B.5). If the instantiated scenario does not satisfy the constraints, the simulator instantiates another scenario until the constraints are met. An example is given below which states that the value of *:destination* and *:source* should not be the same.

```
:scenario_constraint ":destination !:source"
```

:scenario_phase

This key specifies the phases of the scenario. Phases refer to the different stages of the dialogue where a same key in the scenario carries different values. The initial phase is assumed to be **init**. If phases are not specified, the scenario is considered to have only the initial phase.

Every key in the scenario template should specify the phase to associate the correct values with the phases, unless it appears on the "out phase keys" list, in which case it has the same value during all phases. To indicate the phase of the key in the scenario and other places, "%" is used as follows. The initial phase "%*init*" can be omitted.

```
:departure_date%flight2 {c eform ... }
```

Each phase has an entering condition and a finishing condition. Both are logical expression with the same syntax as that of the scenario constraints. The conditions are specified under the key *:scenario_phases*, together with a list of out phase keys. Following is an example which defines two phases. The conditions are tested against the memory state of the simulator.

```
:scenario_phases {c phases
  :out_phase_keys ":itinerary_type"
  :*init* {c phase_cond
        :enter ":flights_len[:itinerary] 0"
        :finish ":flights_len[:itinerary] > 0" }
  :flight2 {c phase_cond
        :enter ":flights_len[:itinerary] 1"
        :finish ":flights_len[:itinerary] !1" } }
```

:scenario_post_process and :registered_ks_handlers

These two keys are used for specifying the post process of the instantiated scenario. The post process calls function handlers that can be found in the KS_HANDLER mappings included in the dialogue domain recipe of this specific domain. All the handlers used in the post processing should also be declared using the key *:registered_ks_handlers*.

The key *:scenario_post_process* takes a list of strings indicating the steps of post process as its value. The steps are carried out in sequence. Each step includes a handler and three parameters. The first parameter specifies the input key. Use *:scenario* if the entire scenario frame is used as the input. The second parameter specifies whether the "session info" frame obtained from previous steps should be send to the handler. The third parameter specifies whether the resultant "session info" frame after calling the handler should be saved for the subsequent steps. An example is given below. In the example, the handler "resolve_date" produces a base date in the "session info" frame. The "session info" frame with a base date is saved after the first step, and is then used in the second step to resolve the departure date of the second flight.

```
:registered_ks_handlers ( "resolve_date" "unsolve_date" )
:scenario_post_process ( "resolve_date(:departure_date, false, true)"
                         "resolve_date(:departure_date%flight2, true, false)" )
```

C.3 Response Strategies

The response strategies include the declaration of personality dimensions, the memory state operation, and the response rules. When the DCTL operation "user_simulation" is executed, or the functions "user_simulation" or "user_suggestions" are called, the simulator generates one or multiple user responses according to FAUNA's reply, the current memory state, and the response strategies.

All of the specifications are written in the strategy frame using a top-level key *:strategies*. Following is an illustrative strategy frame. The four main keys in the frame, :memory, :personality, :condition_penalty and :responses are explained in detail in this section. The specification includes all possible responses. The final responses are chosen by sampling according to the likelihood of each response based on its condition penalty, personality description and the simulated user's personality.

```
:strategies {c strategies
  :memory (
     {c rule
       ... } )
  :personality {c personality
     :cooperativeness {c param
          ... } }
  :condition_penalty ( 0 0 1 1 )
  :responses (
      {c rule
         ... } ) }
```

:memory

The key *:memory* is used to specify the rules for memory state operation. The memory state is a frame maintained for the entire dialogue session to store information received from the dialogue manager. The instantiated scenario is also kept in the memory state under the key *:*scenario**. The memory state is passed in when the functions "user_simulation" or "user_suggestions" are called. Thus, it is possible to include additional content in the memory state to achieve desired simulated user responses.

Each memory rule can have an optional condition to trigger the execution of the rule. The possible operations are: storing elements from FAUNA's reply eform into the memory state, removing elements from the memory state, and incrementing/decrementing counters in the memory state. Below is an example rule, followed by the detailed description of the usage of all the keys.

```
{c rule
   :conditions ":*reply* flight_added"
   :save ":topic"
   :into ":itinerary" }
```

:conditions "*logical_expr*"	Specify the condition to trigger the rule. The syntax of the logical expression is the same as that used in the scenario constraints. The test is conducted against a combined frame of FAUNA's reply message and the current memory state. The combined frame also contains a special key *:*reply**, the value of which is the frame name of the reply message. If multiple messages are presented in the reply eform, the messages are combined with the current memory state one by one, and the rule applies on each of the combined frames.
:save "*:save_key*" :into "*:into_key*"	Append the object specified by *:save* from FAUNA's reply message to the list specified by *:into* in the memory state. If *:into_key* does not exist in the memory state, a

	new list object will be created.
:save "*:save_key*" :as "*:as_key*"	Copy the object specified by *:save* from FAUNA's reply message into the memory state using the key specified by *:as*. If *:as* is not presented, the object will be copied using its original key.
:remove "*:key*"	Remove the key specified by *:remove* from the memory state.
:remove "*:remove_key*" :from "*:from_key*"	Look up the id of the object specified by *:remove* in FAUNA's reply message. If the value of *:from_key* in the memory state has the same id, remove *:from_key* from the memory state; or if the value of *:from_key* in the memory state is a list, and one of the elements in the list has the same id, remove that element from the list.
:increment ":key" :increment (":key1" ":key2" ...)	Increment the integer value of the counter key(s) by 1.
:decrement ":key" :decrement (":key1" ":key2" ...)	Decrement the integer value of the counter key(s) by 1.

:personality

This key is used to specify the personality dimensions of the simulated user. All the dimensions are specified in a frame, in which every key is a personality dimension. In the simulation of a dialogue, each personality dimension will be assigned a fixed or a random value between 0 and 1 according to the specification. Different values will lead to different probability distributions over all the possible responses.

An example of the personality specification is given below, in which one personality dimension "aggressiveness" is defined.

```
:personality {c personality
  :aggressiveness {c param
```

```
        :range ( 0 1 )
        :step_base 10
        :conditions+ ( "#key > 3" )
        :conditions- ( ":command delete" ) } }
```

The meaning of the keys in the example can be found in the following list.

:fixed_value *float*	Assign a fixed value for this dimension. The value should be between 0 and 1.
:range (*float float*)	Specify a range of a random value for this dimension. The range should be within 0 and 1.
:step_base *int*	Specify the step base of this dimension. This number reflects how easy it is to observe this personality dimension from an observer's point of view. A small step base means it is easy to be observed, *i.e.*, in a few responses it can be observed. A large step base means it is hard to be observed, *i.e.* many responses which are not neutral in this dimension are required before it can be observed.
:conditions+ ("logical_expr1" "logical_expr2" ...)	Specify the global positive conditions for this dimension. If the user response satisfies one of the conditions, the response is considered to show a positive sign in this dimension.
:conditions- ("logical_expr1" "logical_expr2" ...)	Specify the global negative conditions for this dimension. If the user response satisfies one of the conditions, the response is considered to show a negative sign in this dimension.

:condition_penalty

This key specifies the four condition penalty coefficients for the response rules. The key takes a list of four real numbers between 0 and 1. The four coefficients correspond to the four test results: not satisfied, no condition, weak condition, and satisfied. See the explanation for *:responses* for detailed information.

:responses

This key is used to specify the response rules. The response rules state all the possible responses under different conditions. The value of this key can be a single response rule frame or a list of response rule frames.

Each rule frame contains a list of templates for the response content. It can optionally contain a condition expression, a list of response commands, and a response action. The condition indicates the most desired situation for these responses. The response commands are essentially the frame name of the response eforms. Each command in the list can be associated with any response content in the rule. Thus, a response rule with *n* response content templates and *m* response commands can result in *m*n* possible responses. The response action specifies special actions to perform to the memory state before instantiating the response content templates. Following is a simple response rule.

```
{c rule
  :conditions ":continuant need_{$attr}"
  :response_content ( {c eform :$attr ":$attr[:*scenario*]" } )
  :response_command ( "show" "" "book" ) }
```

The condition of the rule is specified using the key *:conditions*. The condition logical expression is tested against the combination of FAUNA's reply eform and the simulator's memory state. The frame name of the reply eform can be accessed using the special key *:*reply**. The test result is one of the following four:

- Not satisfied: the condition is not satisfied.
- No condition: the rule does not have a condition.
- Weak condition: the condition is satisfied even when testing against the memory state only, *i.e.*, the condition has nothing to do with FAUNA's reply.
- Satisfied: the condition is not weak and is satisfied.

Based on the test result, a corresponding penalty coefficient is given to all the responses in this rule. The penalty coefficients can be specified using *:condition_penalty* in the strategy frame.

The syntax of the logical expression is similar to that of the memory rule conditions and the scenario constraints, with an extra "matching" syntax. The matching syntax is valid only in an equality test of a string value. A matching variable is recognized by a leading "$" and a pair of curly braces. If the string pattern is matched, the matching variable will be assigned the corresponding value, and can then be used in the response content template. In the example above, if the continuant in FAUNA's reply eform is "need_source", the matching variables *$attr* will be assigned the value "source".

The templates of response contents, specified using the key *:response_content*, are frames with keys that are supposed to appear in the user's response. If the value of a key is a string and starts with a colon, it refers to the value of the corresponding key in the memory state. Brackets are allowed to refer to keys in subframes of the memory state. Keys in the instantiated scenario can be accessed by *:key[:*scenario*]*, as shown in the example above. The values of the keys in the response content templates can also be a special function, *e.g.*, a random integer generation function as follows.

```
{c eform
  :nth "random(1, :npending)" }
```

A special key :*other* is also defined to add "other" keys into the response content. When instantiating, the key :*other* is instantiated into one or more random keys, among which at least one has never appeared in previous user's responses to FAUNA. An example usage is shown below.

```
{c eform
  :*other* ":*other*[:*scenario*]" }
```

The templates may also contain template-specific personality descriptions using the personality dimension as keys, and "+" or "-" as values. The intuitive meaning of the syntax can be interpreted as "this response is more likely to be given by users with/without this particular personality characteristic." The following example response rule contains two response content templates. The second one not only provides the information that FAUNA asks for, but also includes some "other" keys, and is thus labeled as "plus" aggressiveness.

```
{c rule
  :conditions ":continuant need_{$attr}"
  :response_content ( {c eform
                             :$attr ":$attr[:*scenario*]" }
                         {c eform
                             :$attr ":$attr[:*scenario*]"
                             :*other* ":*other*[:*scenario*]"
                             :aggressiveness "+" } )
  :response_command ( "show" "" "book :aggressive+" ) }
```

The response commands are specified by the key *:response_command*, the value of which is a list of strings. The commands are essentially the frame names of response eforms. If the command is an empty string, the frame name remains unchanged, *i.e.*, whatever frame name is written in the response content template is used.

When specifying the commands, personality descriptions can be added using "*:dimension+*" or "*:dimension-*", as shown in the example above. The descriptions both in the content template and in the commands, as well as the satisfaction of any global personality conditions contribute to the likelihood of this response being chosen as the final response.

Some response might require an action before instantiating the content template. In this case, a response action can be specified using the key *:response_action*. Currently, the only implemented action is "change_scenario", which alters the value of one key in the instantiated scenario to reflect a change in the user intention. The following example elucidates the usage of this feature. Personality descriptions are allowed in the action frame as well. If the action fails for any reason, this response would have 0 probability to be chosen as the final response.

```
{c rule
   :response_action {p change_scenario
              :change ":source"
              :aggressiveness "-" }
   :response_content ( {c eform
                 :source ":source[:*scenario*]" } ) }
```

C.4 Other resources

Two example specifications can be found at:

```
$GALAXY_ROOT/galaxy/System/mercury/user_sim.frame
$GALAXY_ROOT/galaxy/System/LanguageLearning/mercury/user_sim_game.frame
```

The second one includes contents specific to the Mercurial dialogue game as well, but can serve as an example of a flexible usage of the user simulator.

The source code of the user simulator can be found at

```
$GALAXY_ROOT/nlcore/src/libdialogue/user_sim.c
$GALAXY_ROOT/nlcore/src/libdialogue/user_sim_int.h
```

References

[1] (2007, Dec.) Xinhuanet. [Online]. http://news.xinhuanet.com/english/2007-12/06/content_7212089.htm

[2] (2011, July) CNTV. [Online]. http://news.cntv.cn/20110721/117162.shtml

[3] (1999) Rosetta Stone. [Online]. http://www.rosettastone.com

[4] (1996) English Town. [Online]. http://www.englishtown.com

[5] (2007) Fluenz. [Online]. http://www.fluenz.com

[6] (2004) Chengo Chinese. [Online]. http://www.elanguage.cn

[7] S. M. Witt and S. Young, "Phone-level Pronunciation Scoring and Assessment for Interactive Language Learning," *Speech Communication*, vol. 30, pp. 95-108, 2000.

[8] W. K. Lo, A. M. Harrison, H. Meng, and L. Wang, "Decision Fusion for Improving Mispronunciation Detection Using Language Transfer Knowledge and Phoneme-Dependent Pronunciation Scoring," in *Proc. ISCSLP*, Kunming, China, 2008.

[9] A. M. Harrison, W.-k. Lo, X.-j. Qian, and H. Meng, "Implementation of an Extended Recognition Network for Mispronunciation Detection and Diagnosis in Computer-Assisted Pronunciation Training," in *Proc. SIGSLaTE*, Warwickshire, UK, 2009.

[10] M. Suzuki, D. Luo, N. Minematsu, and K. Hirose, "Improved Structure-based Automatic Estimation of Pronunciation Proficiency," in *Proc. SIGSLaTE 2009*, Warwickshire, UK, 2009.

[11] S. Bhat, M. Hasegawa-Johnson, and R. Sproat, "Automatic Fluence Assessment by Signal-Level Measurement of Spontaneous Speech," in *Proc. L2WS*, Tokyo, Japan, 2010.

[12] T. Visceglia, C.-y. Tseng, Z.-y. Su, and C.-F. Huang, "Interaction of Lexical and Sentence Prosody in Taiwan L2 English," in *Proc. L2WS*, Tokyo, Japan, 2010.

[13] Y.-B. Wang, H.-M. Wang, and L.-S. Lee, "Virutal Chinese Tutor (VCT) - A Chinese Language Pronunciation Learning Software," in *Proc. SIGSLaTE, Demo Session*, Warwickshire, UK, 2009.

[14] H. Kunichika, A. Takeuchi, and S. Otsuki, "A Multimedia Language Learning Environment with Intelligent Tutor," in *Proc. International Conference on Computers in Education*, Taiwan, 1993.

[15] H. Kunichika, T. Katayama, T. Hirashima, and A. Takeuchi, "Automated Question Generation Methods for Intelligent English Learning Systems and its Evaluation," in *Proc. ICCE2004*, 2003.

[16] T. Kawahara, H. Wang, Y. Tsubota, and M. Dantsuji, "Japanese CALL System Based on Dynamic Question Generation and Error Prediction in ASR," in *Proc. SIGSLaTE, Demo Session*, Warwickshire, UK, 2009.

[17] E. Levin, R. Pieraccini, and W. Eckert, "Learning Dialogue Strategies within the Markov Decision Process Framework," in *Proc. ASRU 1997*, Santa Barbara, USA, 1997.

[18] K. Scheffler and S. Young, "Corpus-based dialogue simulation for automatic strategy learning and evaluation.," in *Proc. NAACL Workshop on Adaptation in Dialogue*, Pittsburgh, USA, 2001.

[19] M. Frampton and O. Lemon, "Learning more effective dialogue strategies using limited dialogue move features," in *Proc. ACL*, Sidney, Australia, 2006, pp. 185 - 192.

[20] J. D. Williams and S. Young, "Partially observable Markov decision processes for spoken dialog systems," *Computer Speech & Language*, vol. 21, no. 2, pp. 393-422, 2007.

[21] O. Lemon and O. Pietquin, "Machine learning for spoken dialog systems," in *Proc. INTERSPEECH 2007*, Antwerp, Belgium, 2007, pp. 2685–2688.

[22] A. W. Black, S. Burger, B. Langner, G. Parent, and M. Eskenasi, "Spoken Dialogue

Challenge 2010," in *Proc. SLT*, Berkeley, CA, USA, 2010.

[23] B. Thomson et al., "Bayesian Dialogue System for the LET'S GO Spoken Dialogue Challenge," in *Proc. SLT*, Berkeley, CA, USA, 2010.

[24] J. D. Williams, I. Arizmendi, and A. Conkie, "Demonstration of AT&T "LET'S GO": A Production-Grade Statistical Spoken Dialogue System," in *Proc. SLT*, Berkeley, CA, USA, 2010.

[25] T. Paek and R. Pieraccini, "Automating spoken dialogue management design using machine learning: An industry perspective," *Speech Communication*, vol. 80, no. 8-9, pp. 716-729, 2008.

[26] F. Ehsani and E. Knodt, "Speech Technology in Computer-Aided Language Learning: Strengths and Limitations of a New CALL Paradigm," *Language Learning & Technology*, vol. 2, no. 1, pp. 54-73, 1998.

[27] F. Ehsani, J. Bernstein, and A. Najmi, "An interactive dialog system for learning Japanese," *Speech Communication*, vol. 30, no. 2-3, pp. 167-177, 2000.

[28] A. Raux, B. Langner, A. W. Black, and M. Eskenazi, "LET'S GO: Improving Spoken Dialog Systems for the Elderly and Non-natives," in *Proc. Eurospeech 2003*, Genewa, Switzerland, 2003, pp. 753-756.

[29] A. Raux and M. Eskenazi, "Using Task-Oriented Spoken Dialogue Systems for Language Learning: Potential, Practical Applications and Challenges," in *Proc. ISCA ITRW INSTiL04*, Venice, Italy, 2004, pp. 147-150.

[30] D. Bohus and A. I. Rudnicky, "RavenClaw: Dialog Management Using Hierarchical Task Decomposition and an Expectation Agenda," in *Proc. Eurospeech*, Geneva, Switzerland, 2003.

[31] S. Seneff, C. Wang, and C.-y. Chao, "Spoken dialogue systems for language learning," in *Proc. HLT-NAACL*, Rochester, NY, USA, 2007.

[32] S. Seneff, R. Lau, and J. Polifroni, "Organization, Communication, and Control in the

GALAXY-II Conversational System," in *Proc. Eurospeech*, Budapest, Hungary, 1999.

[33] S. Seneff, "Response Planning and Generation in the Mercury Flight Reservation System," *Computer Speech and Language*, vol. 16, pp. 283-312, 2002.

[34] V. Zue et al., "JUPITER: a telephone-based conversational interface for weather information," *IEEE Transactions on Speech and Audio Processing*, vol. 8, no. 1, pp. 85-96, 2000.

[35] P. Wik, "Designing a Virtual Language Tutor," in *Proc. The XVIIth Swedish Phonetics Conference, Fonetik 2004*, Stockholm University, 2004, pp. 136-139.

[36] P. Wik, A. Hjalmarson, and J. Brusk, "DEAL, A Serious Game for CALL, Practicing Conversational Skills in the Trade Domain," in *Proc. SIGSLaTE*, Pennsylvania, USA, 2007.

[37] G. Skantze, "Galatea: a discourse modeller supporting concept-level error handling in spoken dialogue systems," in *Proc. SigDIAL 2005*, Lisbon, Portugal, 2005, pp. 178-189.

[38] M. Eskenazi, "An overview of spoken language technology for education," *Speech Communcation*, vol. 51, pp. 832-844, 2009.

[39] Y. Han (韩愈), "On the Teacher (师说)," 768~824, Tang Dynasty.

[40] A. Gruenstein, I. McGraw, and I. Badr, "The WAMI Toolkit for Developing, Deploying, and Evaluating Web-Accessible Multimodal Interfaces," in *Proc. ICMI*, Crete, Greece, 2008.

[41] A. Gruenstein, "Toward Widely-Available and Usable Multimodal Conversational Interfaces," Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, PhD Thesis 2009.

[42] I. McGraw and S. Seneff, "Speech-enabled Card Games for Language Learners," in *Proc. AAAI*, Chicago, USA, 2008.

[43] B. Yoshimoto, I. McGraw, and S. Seneff, "Rainbow Rummy: A Web-based Game for Vocabulary Acquisition using Computer-directed Speech," in *Submitted to SIGSLaTE 2009*, 2009.

[44] I. McGraw, A. Gruenstein, and A. Sutherland, "A Self-Labeling Speech Corpus: Collecting Spoken Words with an Online Educational Game," in *Proc. Interspeech*, Brighton, UK, 2009.

[45] A. Gruenstein, I. McGraw, and A. Sutherland, "A Self-Transcribing Speech Corpus: Collecting Continuous Speech with an Online Educational Game," in *Proc. SLaTE*, Warwickshire, UK, 2009.

[46] (2011) Quizlet. [Online]. http://quizlet.com

[47] S. Seneff et al., "GALAXY-II: A Reference Architecture for Conversational System Development," in *Proc. ICSLP*, Sydney, Australia, 1998.

[48] C. Wang and S. Seneff, "A Spoken Translation Game for Second Language Learning," in *AIED*, Marina del Rey, California, 2007.

[49] C.-y. Chao, S. Seneff, and C. Wang, "An Interactive Interpretation Game for Learning Chinese," in *the SLaTE Workshop*, Farmington, Pennsylvania, 2007.

[50] J. P. Gee, "What video games have to teach us about learning and literacy," *Computers in Entertainment (CIE) - Theoretical and Practical*, vol. 1, no. 1, p. 20, October 2003.

[51] Y. Xu and S. Seneff, "Mandarin Learning Using Speech and Language Technologies: A Translation Game in the Travel Domain," in *Proc. ISCSLP*, Kunming, China, 2008, pp. 29-32.

[52] K. Yamada and K. Knight, "A syntax based statistical translation model," in *Proc. ACL*, Toulouse, France, 2001.

[53] P. Koehn et al., "Moses: open source toolkit for statistical machine translation," in *Proc. ACL, Interactive Poster and Demo Session*, Stroudsburg, PA, USA, 2007.

[54] S. Seneff, "TINA: A Natural Language System for Spoken Language Applications," *Computational Linguistics*, vol. 18, no. 1, pp. 61 - 86, March 1992.

[55] L. Baptist and S. Seneff, "Genesis-II: A Versatile System for Language Generation in

Conversational System Applications," in *ICSLP*, Beijing, China, 2000, pp. 271-274.

[56] Y. Xu, J. Liu, and S. Seneff, "Mandarin Language Understanding in Dialogue Context," in *Proc. ISCSLP*, Kunming, China, 2008, pp. 113-116.

[57] L. Hetherington, "The MIT Finite-State Transducer Toolkit for Speech and Language Processing," in *Proc. ICSLP*, Jeju, South Korea, 2004.

[58] Y. Xu, A. Goldie, and S. Seneff, "Automatic Question Generation and Answer Judging: A Q&A Game for Language Learning," in *Proc. SIGSLaTE*, Warwickshire, United Kingdom, 2009.

[59] Y. Xu, "Combining Linguistics and Statistics for High-Quality Limited Domain English-Chinese Machine Translation," MIT, Cambridge, Massachusetts, Master Thesis 2008.

[60] J. Glass, "A probabilistic framework for segment-based speech recognition," *Computer Speech and Language*, vol. 17, no. 2-3, pp. 137-152, April/July 2003.

[61] W. I. Hallahan, "DECtalk software: Text-to-speech technology and implementation," *Digital Technical Journal*, vol. 7, no. 4, pp. 5-19, 1995.

[62] J. Hochberg, N. Kambhatla, and S. Roukos, "A flexible framework for developing mixed-initiative dialog systems," in *Proc. the 3rd SIGdial workshop on Discourse and dialogue* , Philadelphia, Pennsylvania , 2002, pp. 60-63.

[63] J. Liu and S. Seneff, "Automatic Drug Side Effect Discovery from Online Patient-Submitted Reviews: Focus on Statin Drugs," in *Proc. IMM*, Barcelona, Spain, 2011.

[64] J. Liu and S. Seneff, "A Dialogue System for Accessing Drug Reviews," in *Proc. ASRU*, Hawaii, USA, 2011.

[65] W. Eckert, E. Levin, and R. Pieraccini, "User Modeling For Spoken Dialogue System Evaluation," in *Proc. ASRU*, Santa Barbara, USA, 1997, pp. 80-87.

[66] L. Esther, P. Roberto, and W. Eckert, "A stochastic model of human-machine interaction for learning," *IEEE Trans. on Speech and Audio Processing*, vol. 8, no. 1, pp. 11-23, 2000.

[67] K. Scheffler and S. Young, "Probabilistic simulation of human-machine dialogues," in *Proc. ICASSP*, Istanbul, Turkey, 2000, pp. 1217-1220.

[68] K. Scheffler, "Automatic design of spoken dialogue systems," Cambridge University, PhD Thesis 2002.

[69] O. Pietquin and T. Dutoit, "A probabilistic framework for dialog simulation and optimal strategy learning," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 14, no. 2, pp. 589 - 599, 2006.

[70] H. Cuayahuitl, S. Renals, O. Lemon, and H. Shimodaira, "Human-computer dialogue simulation using hidden Markov models," in *Proc. ASRU 2005*, San Juan, Puerto Rico , 2005, pp. 290-295.

[71] Y. Xu and S. Seneff, "A Generic Framework for Building Dialogue Games for Language Learning: Application in the Flight Domain," in *Proc. SLaTE*, Venice, Italy, 2011.

[72] M. Gabsdil and O. Lemon, "Combining Acoustic and Pragmatic Features to Predict Recognition Performance in Spoken Dialogue Systems," in *Proc. ACL*, 2004.

[73] O. Lemon and I. Konstas, "User Simulations for context-sensitive speech recognition in Spoken Dialogue Systems," in *Proc. European Chapter of ACL*, Athens, Greece, 2009.

[74] A. Gruenstein, "Response-Based Confidence Annotation for Spoken Dialogue Systems," in *Proc. SIGDial*, Columbus, Ohio, USA, 2008.

[75] T. Joachims, "Making large-Scale SVM Learning Practical.," in *Advances in Kernel Methods - Support Vector Learning*, Alexander J. Smola, Ed.: MIT-Press, 1999.

[76] M. Rayner, N. Tsourakis, P. Bouillon, and M. Fuchs, "CALL-SLT/Web, A Speech-Enabled Translation Game on the Internet," in *Proc. SLaTE, Demo Session*, Tokyo, Japan, 2010.

[77] M. Rayner et al., "CALL-SLT: An Automatic Speech-Enabled Conversation Partner on the Web," in *Proc. SLaTE, Demo Session*, Venice, Italy, 2011.

[78] A. Goldie, "CHATTER: A Spoken Language Dialogue System for Language Learning Applications," Department of Electrical Engineering and Computer Science, MIT,

Cambridge, MA, USA, M. Eng Thesis 2011.

[79] W. Ling, I. Trancoso, and R. Prada, "An Agent Based Competitive Translation Game for Second Language Learning," in *Proc. SLaTE*, Venice, Italy, 2011.

[80] I. McGraw, B. Yoshimoto, and S. Seneff, "Speech-enabled Card Games for Incidental Vocabulary Acquisition in a Foreign Language," *Speech Communication*, 2008.

[81] Z. Yang et al., "Collection of User Judgments on Spoken Dialogue System with Crowdsourcing," in *Proc. SLT*, Berkeley, CA, USA, 2010.

[82] M. Rayner, I. Frank, C. Chua, N. Tsourakis, and P. Bouillon, "For a Fistful of Dollars: Using Crowd-Sourcing to Evaluate a Spoken Language CALL Application," in *Proc. SLaTE*, Venice, Italy, 2011.