

**A Distributed Architecture for Interactive  
Robots Based on  
a Knowledge Software Platform**

Pattara Kiatisevi

DOCTOR OF  
PHILOSOPHY

Department of Informatics,  
School of Multidisciplinary Sciences,  
The Graduate University for Advanced Studies (SOKENDAI)

2005 (School Year)

September 2005

A dissertation submitted to the Department of Informatics,  
School of Multidisciplinary Sciences,  
The Graduate University for Advanced Studies (SOKENDAI)  
in partial fulfillment of the requirements for  
the degree of Doctor of Philosophy.

Advisory Committee:

Advisor	Prof. Haruki Ueno	National Institute of Informatics (NII), SOKENDAI
Sub-advisors	Assoc. Prof. Tomoko Matsui	Institute of Statistical Mathematics
	Assoc. Prof. Nigel Collier	NII, SOKENDAI
Members	Assoc. Prof. Nobuhito Furuyama	NII, SOKENDAI
	Prof. Shin'ichi Satoh	NII, SOKENDAI
	Prof. Yoshiaki Shirai	Osaka University

# Abstract

Robots have taken more and more important roles in human life. That is, they are not only working in factories and hazardous environment but also being introduced in applications like welfare services and amusement agents, where interaction with human is crucial. In the future, we envision interactive robots assisting humans especially the elderly and the disabled in daily life. Such robots coexist and interact with humans in a human-friendly manner.

To develop such robots we need to address challenging problems of how to integrate various robotic components, and how to control robot behaviors so that it interacts with humans reasonably. The first problem involves architectural design of the system in the component level, i.e., how various types of robotic components can be combined into a functional integrated system. The second problem involves designing the brain part of the system, which manages robot actions and interactions with human. There are several approaches in designing robot architectures to accommodate these problems, e.g., behavior-based, plan-based, and knowledge-based approaches. In our research group, we have been focusing on the knowledge-based approach with the emphasis on the frame model, and have developed a frame-based software platform for robots called the *Software Platform for Agents and Knowledge* or SPAK.

In this work, we extended SPAK to overcome the problems of representing changing knowledge and limitations in managing human-robot interactions. New extensions to the conventional frame model, namely, time-based layer, periodical task evaluator, and priority support for frame-related actions, are proposed and added to SPAK. With the time-based layer, the SPAK-based knowledge manager keeps track of changes in the knowledge contents and provides methods to access this history data. One can, for example, query frames' age, old values of slots at absolute or relative points in time, and specify frames' condition based on this history data. The periodical task evaluator regularly checks the validity of the knowledge contents, fixes it if needed, and executes actions. Designers can specify actions to be done periodically in a special event-driven slot whose contents will be executed by the evaluator. Priority support for frame-related actions gives the knowledge designer a control over the execution order of frame-related actions. Each frame can be assigned a priority value. The inference engine in SPAK will respect these values when conducting actions like

frame instantiation, updating, deactivation, and evaluation. These new extensions can be used via new special system and event-driven slots, and slot flags.

To verify the utility of SPAK and to illustrate how a robot system can be built based on this platform, a demonstration system on a humanoid robot was developed, with a sample dialogue management application. The knowledge model for the dialogue manager is shown. The current knowledge-based dialogue manager can handle basic human commands, state-based and form-based dialogues. A multi-agent technique is employed to connect various robotic hardware and software components designed as agents together on the network. The prototype system was set to make dialogue interactions with a human in three scenarios: basic learning, greeting, and future welfare robot.

This dissertation makes three contributions to the fields of knowledge engineering, dialogue systems, and robotics.

First, the thesis proposes a design concept for the knowledge manager, the brain part of the robot. We based our design on the frame knowledge model because of its simplicity and naturalness, and introduced three dynamic extensions to the conventional frame model to support robot behaviors control. Various robotic applications can run and share knowledge among each other on the platform.

Second, the thesis proposes a novel design of a knowledge-based dialogue manager for robot as an application on the platform. We show that the design and the internal mechanism of the system are natural and easy for developers to understand. With tight integration to the knowledge base, the dialogue manager can easily make use of the knowledge and knowledge-related facilities provided.

Third, the thesis contributes a knowledge-based robot architecture with an implementation of a robot system that interacts with humans in the laboratory environment. The system components include SPAK, the designed dialogue manager running in SPAK, various robot software and hardware components, and a Robovie humanoid robot. With SPAK, we show that it is simple and intuitive to develop a multi-modal interactive robot system.

A SPAK-based robot system has an important feature of frames and agents being integrated seamlessly in a distributed environment. A prototype system with a sample dialogue management application has demonstrated interesting functions for future symbiotic robots. Ultimately, this dissertation demonstrates the concept of employing a knowledge platform as the base layer of a robot system. The platform works as glue connecting various robotic devices and applications together. We believe that this is a major step towards achieving intelligent future robots.

# Acknowledgement

I would like to thank many people for their support, encouragement and guidance during my years as a graduate student at Sokendai/NII.

First and foremost, this dissertation represents a great deal of effort not only of mine, but also my advisor, Haruki Ueno. He has helped me shape my research from day one, guided and pushed me to get through all the inevitable research setbacks. Without him, this dissertation would not have happened.

I also thank my sub-advisors Tomoko Matsui, Nigel Collier, and the committee members, Nobuhiro Furuyama, Shin'ichi Satoh, Yoshiaki Shirai, and Seiji Yamada and other professors at NII, for valuable discussions and comments regarding my research.

This research is indeed a group effort. I deeply thank Vuthichai Ampornaramveth for his valuable advice and supports to making my research possible. I also thank other Ueno laboratory members, Md. Hasanuzzaman, Alexander Kovacs, Chikahito Nakajima, Zhang Tao, for helps, valuable and fruitful discussions in the "Friday" meetings and during the course of my research.

My life at NII would not have been smooth without great support from many NII staff. I thank all of them, especially Tami Nakata, Miyoko Taylor, Nahoko Iwanaga, Sayo Eibahara, Aiko Kawamura, and the previous members of NII: Fuyuki Matsushita, Yumi Imai, and Aiko Uchida.

On a more personal note, I have been lucky to have many close friends and classmates, especially my fellow students at NII, who have made my times as a graduate student in Japan very enjoyable. I specially thank Vuthichai Ampornaramveth (again) and Panrit Tosukowong for their great supports of my living in Japan since the very first days.

Most of all, I would like to thank my mother, father, and brother, who have always been and continue to be there for me all the time. And lastly Supavadee Monsathaporn, for your patience and supports during my years in Japan.

27 September 2005

Pattara Kiatisevi



# Contents

<b>Abstract</b>	<b>3</b>
<b>Acknowledgement</b>	<b>5</b>
<b>Contents</b>	<b>7</b>
<b>List of Figures</b>	<b>10</b>
<b>List of Tables</b>	<b>13</b>
<b>1 Introduction</b>	<b>15</b>
1.1 Motivation . . . . .	15
1.2 Objectives and Approach . . . . .	18
1.2.1 Thesis Question . . . . .	18
1.2.2 Approach . . . . .	18
1.3 Contributions . . . . .	20
1.4 Organization of the Thesis . . . . .	21
<b>2 Background</b>	<b>22</b>
2.1 Interactive Robot Architecture . . . . .	22
2.1.1 Architecture Design Approaches . . . . .	23
2.1.2 Distributed Systems . . . . .	25
2.1.3 Interactive Robot Systems . . . . .	27
2.1.4 Concerned Difficulties and Issues . . . . .	28
2.2 Frame-based Knowledge Systems . . . . .	28
2.2.1 Frame Knowledge Model . . . . .	29
2.2.2 Frame-based systems . . . . .	31
2.2.3 Applications of Frame-based systems in Robotics . . . . .	32
2.2.4 Concerned Difficulties and Issues . . . . .	34
2.3 Dialogue Systems . . . . .	35
2.3.1 Dialogue Systems Components . . . . .	36

2.3.2	Dialogue Management Approaches . . . . .	37
2.3.3	Dialogue Systems . . . . .	42
2.3.4	Frame-based Knowledge Technique and Dialogue Systems . . . . .	46
2.3.5	Concerned Difficulties and Issues . . . . .	46
<b>3</b>	<b>Knowledge-based Distributed Robot Architecture</b>	<b>48</b>
3.1	Introduction . . . . .	48
3.2	System Architecture . . . . .	49
3.2.1	Primitive Agent . . . . .	49
3.3	Agents Collaboration . . . . .	51
3.4	Knowledge Manager Agent . . . . .	52
3.5	Technical Design . . . . .	53
3.5.1	Communications among Primitive Agents . . . . .	54
3.5.2	Primitive Agent Abstraction . . . . .	55
3.6	Summary . . . . .	56
<b>4</b>	<b>Frame-based Knowledge Manager</b>	<b>57</b>
4.1	Frame Model . . . . .	57
4.2	Knowledge Manager Roles . . . . .	58
4.3	Frame Model Extensions . . . . .	59
4.3.1	Special Slots . . . . .	60
4.3.2	Slot Flags . . . . .	63
4.3.3	Time-based Layer . . . . .	65
4.3.4	Evaluator . . . . .	66
4.3.5	Priority Support for Frame Actions . . . . .	67
4.4	SPAK Knowledge Platform . . . . .	68
4.4.1	Graphics User Interface (GUI) . . . . .	69
4.4.2	Knowledge Base . . . . .	69
4.4.3	Inference Engines . . . . .	75
4.4.4	JavaScript Interpreter . . . . .	76
4.4.5	Network Gateway . . . . .	76
4.5	SPAK Reasoning Mechanism . . . . .	77
4.5.1	Scene Understanding . . . . .	80
4.5.2	Robotic Tasks Planning . . . . .	82
4.6	SPAK Knowledge Design Policy . . . . .	82
4.7	SPAK Programming Interfaces . . . . .	88
4.8	Summary . . . . .	89



<b>5</b>	<b>SPAK Application: Knowledge-based Dialogue Manager</b>	<b>90</b>
5.1	Design of the Dialogue System for Robots . . . . .	90
5.2	Knowledge-based Dialogue Manager . . . . .	92
5.3	Dialogue Management . . . . .	98
5.3.1	Handling of Human Commands . . . . .	99
5.3.2	Asking a Question . . . . .	100
5.3.3	Handling of State-based Dialogues . . . . .	103
5.3.4	Handling of Form-based Dialogues . . . . .	103
5.3.5	Robot Learning from Human Instruction . . . . .	105
5.4	Summary . . . . .	106
<b>6</b>	<b>Prototype Development</b>	<b>109</b>
6.1	Robot Components . . . . .	109
6.2	Interaction Scenarios . . . . .	114
6.3	Dialogue Manager Internal Mechanisms . . . . .	118
6.4	Discussions . . . . .	120
6.5	Summary . . . . .	121
<b>7</b>	<b>Evaluation</b>	<b>122</b>
7.1	SPAK Extended Frame-based System . . . . .	122
7.2	Knowledge-based Dialogue Manager . . . . .	125
7.3	Distributed Knowledge-based Robot Architecture . . . . .	126
<b>8</b>	<b>Conclusions and Future Work</b>	<b>130</b>
8.1	Contributions . . . . .	130
8.2	Limitations and Future Directions . . . . .	131
8.3	Concluding Remarks . . . . .	132
	<b>About Author</b>	<b>133</b>
	<b>Related Publications</b>	<b>134</b>
	<b>Bibliography</b>	<b>136</b>
	<b>Index</b>	<b>146</b>

# List of Figures

1.1	Platform-based approach for realizing symbiotic robots . . . . .	19
2.1	Deliberative architecture . . . . .	24
2.2	Reactive architecture . . . . .	24
2.3	Example of a simplified image in identifying a cup vessel using the knowl- edge in the Cup_Vessel frame in the Shape model inside the world model . .	34
2.4	Classical pipe-line structure of a spoken dialogue system . . . . .	37
2.5	Multimodal dialogue systems have more modes of input/output, . . . . .	38
3.1	Robot as a network of primitive agents: speech recognizer, speech synthesizer, knowledge manager, gesture recognizer, face recognizer, face detector, and robot posture . . . . .	50
3.2	Platform approach for robot architecture . . . . .	51
3.3	Four types of collaboration among agents that compose a robot (the big rect- angle). . . . .	52
3.4	Knowledge manager's knowledge model and interactions with other compo- nents . . . . .	53
4.1	Hierarchy of frames representing lines including an instance <i>Line_1</i> as a child of both <i>LongLine</i> and <i>ThickLine</i> frames. In each frame and instance, its slots are shown in the left column and their values in the right column, optionally with a condition in the square brackets (e.g., the <i>thickness</i> slot must be greater than 2 for the <i>ThickLine</i> frame). Note that the green tables represent frames and the red ones represent instances. Red arrows are used to indicate IS-A relationships.	58
4.2	Input, output, and actions mechanisms of SPAK knowledge manager . . . . .	59
4.3	Snapshots of the frame instance representing <i>John</i> at various time points . . .	66
4.4	A screenshot of SPAK loaded with the knowledge contents representing line frames from the example in Figure 4.1 . . . . .	71

4.5	A screenshot of a property window of the <i>Line_1</i> instance in Figure 4.4. Each row shows a slot with its name, type, value (frame default value), slot condition, argument (of the condition), and slot flags (R, BO, S, U). . . . .	72
4.6	A screenshot of SPAK showing the knowledge of the <i>Human</i> frame and its children . . . . .	73
4.7	Hierarchy of frames representing humans. The red lines represent HAS_A relationships. . . . .	74
4.8	Property of the <i>Child_1</i> instance . . . . .	74
4.9	A flowchart showing algorithm of the SPAK induce and reInduce inference processes . . . . .	78
4.10	A flowchart showing algorithm of SPAK induce and reInduce inference process	79
4.11	Scene understanding with image processing modules and SPAK . . . . .	80
4.12	A frames hierarchy corresponding to the knowledge contents shown in Figure 4.4. The red and blue lines represent IS_A and HAS_A relationships respectively	81
4.13	Task scheduler for move-a-cup action . . . . .	82
4.14	A knowledge hierarchy showing the <i>SeenHuman_1</i> instance and its parents . .	83
4.15	Property of the <i>SeenHuman_1</i> instance . . . . .	84
4.16	A knowledge hierarchy starting from the <i>SeenObject</i> node . . . . .	85
4.17	A knowledge hierarchy starting from the <i>Human</i> node . . . . .	86
5.1	An overview of the designed dialogue manager. A robot is composed of various agents including the SPAK knowledge-manager agent. . . . .	91
5.2	A screen-shot of SPAK showing groups of knowledge frames: <i>StatusRegister</i> , <i>Event</i> , <i>Action</i> , world knowledge ( <i>Object</i> and <i>Concepts</i> ), that constitute the dialogue manager for robots. Note that the frames further down the hierarchies are not shown. . . . .	93
5.3	A SPAK screenshot showing <i>Event</i> and <i>Action</i> frames . . . . .	94
5.4	<i>StatusRegister</i> frames and <i>Objects</i> frames. Note some multiple-parent frames that are children of both the <i>Object</i> and <i>StatusRegister</i> frames . . . . .	95
5.5	A snapshot of the LSE and LSA frames . . . . .	96
5.6	Window function used to calculate the significance factor . . . . .	97
5.7	<i>Concepts</i> frames, as a part of the world knowledge, represent conceptual knowledge like gestures and speech . . . . .	98
5.8	Example of the knowledge contents for processing robot commands, and the content of the <i>SayWhatHumanSaid</i> frame . . . . .	99
5.9	A screenshot of a <i>DialogueAsk_1</i> instance . . . . .	100
5.10	A screenshot of a <i>WaitingForAnswer_1</i> instance created by the <i>DialogueAsk_1</i> instance in Figure 5.9 . . . . .	101

5.11	A screenshot of an <i>AnswerFound</i> frame used to match between a <i>SpeechRecognized</i> instance representing incoming speech from human and a <i>DialogueAsk</i> instance . . . . .	101
5.12	Related frames to the mechanism of asking a question and their interactions . . . . .	102
5.13	Example knowledge contents to realize state-based dialogues . . . . .	103
5.14	A flow chart showing a task action flow, dialogue state frame updates, and interactions with human, in a state-based dialogue for greeting a known user . . . . .	104
5.15	Related frames to the bus reservation dialogue and their interactions . . . . .	107
5.16	Property of a <i>ReserveBus</i> frame, which is an example of a frame-based dialogue with the goal to reserve a bus seat . . . . .	108
6.1	The prototype system . . . . .	109
6.2	System configuration of the prototype system. Agents in the demonstration prototype run on five different machines and communicate on a TCP/IP network using the XML-RPC protocol. . . . .	110
6.3	Menu selection for starting, testing and killing agents running on the Robovie robot . . . . .	114
6.4	Human robot interaction in the scenarios . . . . .	115
6.5	Steps showing changes in SPAK knowledge base during the human-robot interaction scenario . . . . .	118
6.6	A SPAK window showing property of the <i>FoundHuman</i> state frame. . . . .	119
6.7	Snapshot of a <i>GreetNewUserTaskAction1_1</i> instance . . . . .	119
6.8	A SPAK window showing property of the <i>DialogueAsk_1</i> instance during the interaction. It asks human according to the text in its <i>question</i> slot ( <i>Hello, we haven't....</i> ) and creates a <i>WaitingForAnswer</i> instance with the expected speech act according to its <i>answeract</i> and <i>answersubact</i> slots. . . . .	120
7.1	Blackboard-based approach (right) to develop robots where a central brain is employed, compared to the conventional mesh design (left) . . . . .	128

# List of Tables

1.1	Problems and proposed solutions in this work . . . . .	20
2.1	Description of a Red-cup frame . . . . .	33
2.2	Description of a Cup_Vessel frame . . . . .	34
2.3	An example of a frame representing a red cup in the functional model . . . .	34
2.4	Comparison of several dialogue management approaches . . . . .	41
3.1	Example of an XML-RPC request to a remote function <i>ping</i> with one integer parameter . . . . .	55
4.1	List of special slots supported in SPAK . . . . .	60
4.2	List of slot flags in SPAK . . . . .	64
4.3	Supported slot types in SPAK . . . . .	69
4.4	Some methods to manipulate frames provided by the <i>KFrame</i> Java class . . .	87
4.5	Some properties and methods of the <i>KFrameScript</i> class . . . . .	88
7.1	Comparison of SPAK to other frame-based systems . . . . .	124
7.2	Comparison of the knowledge-based dialogue management with the conven- tional dialogue management approach . . . . .	127



# Chapter 1

## Introduction

Robots have taken more and more important roles in human life. That is, they are not only working in factories and hazardous environment but also being introduced in applications like welfare services and amusement agents, where interaction with human is crucial. We human need some helps in daily life. However, many countries including Japan [1] are experiencing aging society, where the average age of population is increasing. In this situation, it is more difficult to receive helps from other humans such as care takers, nurses. One solution is to utilize robots and Information Technology (IT). In the future, we envision interactive robots assisting humans especially the elderly and the disabled in daily life. The robots coexist and interact with humans in a human-friendly manner.

This research is focused on the design and development of such interactive robots. A knowledge-based robot architecture is proposed, with a sample application and a prototype system. This chapter gives an overview of this research, including motivation, objectives, approach, and contributions.

### 1.1 Motivation

Despite the explosive development of IT toward the 21st century, instant adoption of such technologies into everybody's daily life is still not possible. Although recent development in information technology has led to more powerful and useful information systems, they also became more complicated. Access to recent technologies requires some kinds of training for which not everybody can afford. In order to maintain the citizens' "right of usage" of any advanced information technologies, it is necessary to realize an intelligent information system that is accessible by anybody, regardless of his/her education background or age, anywhere, and anytime. We believe that instead of training human users to be computer-literate in order to use the technology, the technology itself should be made intelligent enough to communicate and understand users in a human-friendly way. This is the motivation of NII's research program on the topic of *Symbiotic Information Systems* (SIS) [2, 3, 4].

Symbiotic information system is an information system that includes human beings as an element, blends into human daily life, and is designed on the concept of symbiosis. Development and realization of such a barrier-free system are the long-term objectives of SIS. Researches on SIS cover a broad area including intelligent human-machine interfaces with vision, speech, natural language, virtual reality, agent technologies, ubiquitous computing, network system with guaranteed quality of service, and development of supporting runtime environment. One of our final goals is to achieve *symbiotic robots* which live together with and assists humans, especially the elderly and ill persons, in daily life.

The term *symbiosis* is also used by other researchers and communities for describing relationship between human and robots, e.g., in [5, 6]. Matsuyama views man-machine symbiotic systems as a step further beyond conventional machine interfaces [7]. Instead of the order-response or master-slave model where human acts as an omnipotent master and a machine as an obedient server, in man-machine symbiotic systems, a human and a machine are considered to be on equal terms and interact with each other as partners. The machine should be realized such that it works for humans even if they are not explicitly ordered. They are focusing on topics like human-robot multi-modal interaction, sensory technologies to observe human activities, multimedia presentation methods to attract human, and real-time interaction protocol to interact with humans.

The SIS project of NII has a goal to realize information systems that coexist with humans and interact with us in the human-friendly manner. We aim to achieve symbiotic robots, i.e., the robot based on the concept of SIS that coexists with human at the places of everyday life, communicates with, and assists human. It is foreseeable that such a robot would be helpful in aged society, such as Japan, where there will not be enough younger generations to take care of the aged members. Demand for this kind of care-taking robot is increasing in high-welfare society of the 21st century. Also intelligent service robots which can execute works as proxy under dangerous environment are expected. By combining with the Internet technology these robots become networked intellectual sensors that can not only help people locally, but also expand the welfare service area by cooperating with the welfare center where welfare-service professionals are supporting the total service [8].

Inspired by the above concept, this research is focused on the realization of symbiotic robots. Research in robotics has been progressing rapidly during the last decade. Significant achievement has been made in the mechanical or sensory-and-motor worlds. The robots can now imitate human-biped movement. However, most robots developed so far still lack the ability to interact with human user in natural and ways and act intelligently. There are several issues to consider when developing robots.

Human-robot interaction is one of the most important issues in realizing symbiotic robots. There are many possible human-robot interaction modes— e.g., using a monitor, keyboard and mouse, or more natural means like speech, vision, and gesture— depending on the tar-



get application and the person whom it interacts with. Consider these target users of elderly and disabled persons, it would be inconvenient to ask them to use the keyboard or mouse in order to communicate with machines as they might not be familiar with computer interfaces. Multi-modal natural interfaces like speech and gesture is preferred. Hence, the robot is required to understand human requests in a natural manner, to recognize environment by means of multiple sensors, to respond to change of the environment, to talk with him/her, and so on. We regard this targeted type of robot as *socially-interactive robots* (similar to the definition in [9]), in short, interactive robots.

Interactive robots can have different hardware and software configurations, depending on target capabilities. Consider a robot for household uses, its hardware can include moving devices (e.g., wheel, legs), display devices (e.g., screen, projector, its arms and bodies for displaying gestures), object handling devices (e.g., arm, hand, manipulator, carrier), and processing devices. Also future service robots should be able to work in the networked environment, where they collaborate with other systems such as a central welfare service center over the Internet to serve humans at home. How to combine these components to an integrated robot system is an issue.

A robot equipped with proper sensor and actuator devices is then expected to have some capabilities. Physical capabilities are, for example, moving, navigating, talking, grasping and lifting things. Software or computing capabilities are, e.g., speech recognition, face recognition, scene understanding. Based on these capabilities, the robot exhibits some behaviors to provide some services or accomplish some tasks, which we call it a *robot application*. Example of robot applications are dialogue management (which can act also as a front-end to other applications), security monitoring (e.g. monitor a house and alert if there is an intruder), fetching and delivering things, room or space exploring.

It would be straightforward if each robot application is independent and does not involve others. However, robot applications need to share robot devices and capabilities. In order to achieve consistent behaviors from all applications, they need to share the knowledge they gained, e.g., location knowledge from room explorer, human-related information from dialogue management. To realize this, there must be something that glues every parts together. In other words, the robot architecture must allow integration of various robotic devices and applications and let them work cooperatively. Robot applications should be able to cooperate and share the common knowledge.

The question is how do we design such architecture?, what technologies are needed in the system to fulfill such requirements? In this thesis, we propose the concept of a distributed robot architecture based on a knowledge software platform. To illustrate how a robot system can be built using this concept, a demonstration system on a physical robot is developed. Dialogue management is selected as a sample application on the system as it is a crucial function of symbiotic robots.

## 1.2 Objectives and Approach

In this section we outline the thesis question and the approach used in this work.

### 1.2.1 Thesis Question

The principal question addressed in this thesis is:

**What is a suitable robot architecture for socially-interactive robots that integrates various robotic devices and applications, and allows intuitive development process of and collaboration among various robot applications?**

More specifically this thesis contributes a robot architecture for interactive robots that can:

- integrate various robotic sensor and actuator components, e.g., head, body, cameras, microphone, speakers;
- integrate various robotic software components, e.g., image processing software, speech processing software;
- maintain the robot knowledge about the world of interest, which changes over time;
- manage robot behaviors including the dialogue interaction with humans

### 1.2.2 Approach

To solve the thesis question, we took the *platform approach* based on multi-agent and knowledge techniques. We argue that the combination of knowledge techniques with multi-agent technology on a single general-purpose platform is needed to meet these requirements. Multi-agent technique allows developers to break down the system into independent but collaborative components. The system is thus modular. Development of each components can progress independently and re-using the existing technologies is easy, provided that abstraction of these components is well-defined. Managing human-robot interactions using natural interfaces like speech requires machines to process symbols such as words and sentences used by human, e.g., in natural language speech, as well. We human-beings use symbols as well as knowledge related to them in understanding such as speech and gestures, and in planning actions based on understanding. Therefore the use of symbol-based techniques like logic and other knowledge techniques is beneficial. Lastly there is a problem of integration. Some autonomous software systems have an architecture that allows interconnection of several software modules and results in a too complicated system structure. Addition of new functions is difficult. Instead, we propose to integrate all robotic modules using multi-agent techniques on a networked knowledge-based platform. The development

of such a new general-purpose platform is needed for developing interactive robots. Figure 1.1 depicts the concept of our platform-based approach to developing symbiotic robots. A symbiotic robot is a fusion of various hardware and software functions on the platform.

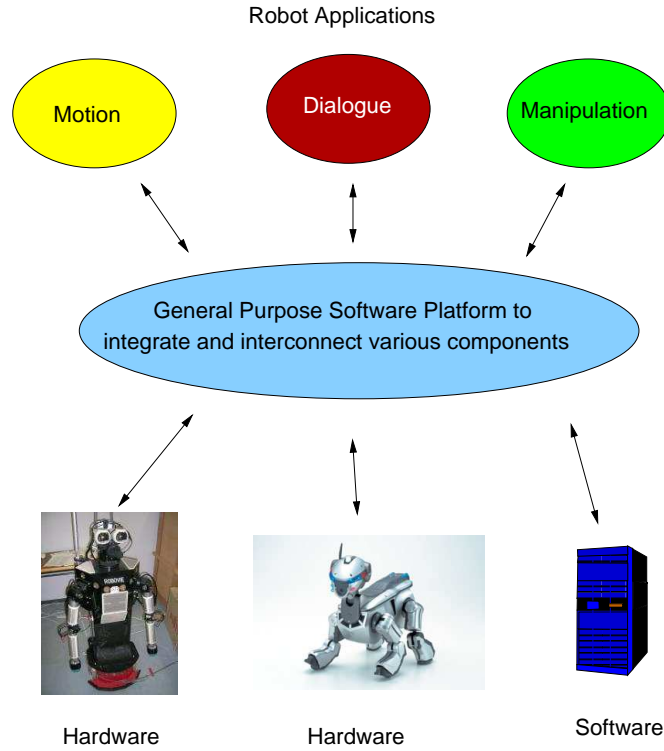


Figure 1.1: Platform-based approach for realizing symbiotic robots

In our design, a robot is composed of networked agents. A special agent knowledge manager, which features a knowledge base, inference engine, and newly added dynamic extensions, manages the robot's actions and interactions with human and environment. We emphasize on the knowledge platform approach and have developed a general purpose knowledge software platform called *Software Platform for Agents and Knowledge Management* or SPAK [10]. Instead of having a dedicated system for each robot application, the platform serves as basis infrastructure for all, ensuring interoperability among them and easy sharing of knowledge.

To represent knowledge in our system, the frame knowledge presentation ("Frame" for short) [11] is used. Frame is preferred over other techniques (e.g., first-order logic) because it can represent the world in a meaningful and natural way. Frame is cognitively simple, intuitive and understandable for domain experts. Limited expressiveness of frame enables tractable inferencing [12]. Frame has been used in many researches, including in robotic systems, e.g., [13]. Uses of frame in other systems are mostly limited to representing knowledge. However, in our system, frame is used both for representing knowledge and for managing

Problems	Proposed Solutions
Representing changing knowledge and management of robot behaviors	Extensions to the frame model and the frame-based knowledge manager (Chapter 4)
A show-case multi-modal robot application	A knowledge-based dialogue system (Chapter 5)
Verification of the proposed concepts	A demonstration prototype on a humanoid robot (Chapter 6) with a distributed multi-agent robot architecture (Chapter 3)

Table 1.1: Problems and proposed solutions in this work

robot behaviors.

In this work, we extended SPAK to overcome the problems of representing changing knowledge and limitations in managing human-robot interactions found when deploying SPAK in a robot system. New extensions to the conventional frame model, e.g., time-based layer, periodical task evaluator, are proposed to solve the issues. To verify the utility of SPAK in an interactive robot system, a sample multi-modal dialogue management application is designed and developed. A prototype system on a humanoid robot was also built and set up to interact with a human in three interaction scenarios. Table 1.1 summarizes the concrete problems and solutions proposed in this thesis.

The contents of this thesis covers the proposed distributed architecture, the extended frame model and the knowledge platform, a sample dialogue management application, and the development of a prototype system on a humanoid robot.

As a conceptual note, the term robot in this work is considered as an engineering device, not in the sense of artificial creatures with evolving general intelligence. Although it should have some learning abilities to improve itself in order to to serve human better, emerging behaviors that can not be explained how and why they happened are not desired.

### 1.3 Contributions

This dissertation makes three distinct contributions to the fields of knowledge engineering, dialogue systems, and robotics.

- **Extended Frame-based Knowledge Model:** we propose dynamic extensions to the conventional frame model and realize it on our in-house SPAK software platform.
- **Knowledge-based Dialogue Manager:** based on the proposed robot architecture and SPAK, a novel knowledge-based dialogue manager is designed and developed.

- **Multi-Agent Knowledge-based Robot Architecture:** we propose an architecture for interactive robots and show a working implementation on a humanoid robot system. The design is based on multi-agent and knowledge platform.

## 1.4 Organization of the Thesis

Following is a general description of the organization of this thesis. Chapter 2 provides background information and reviews of researches in related fields. Chapter 3-6 present the core of this work: the robot architecture, the knowledge manager, the dialogue manager, and their implementations in a prototype system. Chapter 7-8 evaluate and conclude this work. More details of contents in each chapter are shown as follows:

- **Chapter 2 Background:** provides background information and recent progresses in the research of interactive robot architectures and systems, knowledge-based systems, and dialogue systems, with the focus on dialogue system architectures and dialogue management approaches.
- **Chapter 3 Knowledge-based Distributed Robot Architecture:** gives an overview of the proposed robot architecture. Overall architecture design, system structure, agents, and interaction among agents are described.
- **Chapter 4 Frame-based Knowledge Manager:** introduces the design concept of the knowledge manager, the brain part of the robot. First the frame knowledge model is introduced and later we described our dynamic extensions we proposed to the conventional frame model to support robot actions management. Implementation of the proposed concepts in SPAK follows in this chapter.
- **Chapter 5 SPAK Application: Knowledge-based Dialogue Manager:** introduces a knowledge-based dialogue manager as a sample application on the above proposed platform. Concepts and design of the dialogue manager is present and contrasted with other dialogue manager approaches.
- **Chapter 6 Prototype Development:** describes technical details of prototype development and the output system. Development of SPAK-based dialogue manager on a humanoid robot is discussed and the result human-robot interaction in three sample scenarios is shown.
- **Chapter 7 Evaluation:** evaluates the work in thesis and compares it with related work.
- **Chapter 8 Conclusions and Future Work:** presents conclusions and future work.

## Chapter 2

# Background

In this Chapter we provide the background knowledge and a review of the state-of-the-art in three related fields: robotics, knowledge engineering, and dialogue systems technology. The focus is on three main topics: interactive robot architecture, knowledge-based systems and dialogue systems.

### 2.1 Interactive Robot Architecture

The term robot was coined in 1921 by a Czech writer Karel Capek ("Robot" in Czech comes from the word "robota", meaning "compulsory labor"). Nowadays robot generally refers to an entity that performs certain tasks automatically. The term robot usually implies a certain level of autonomy and intelligence.

Robots have been used for decades in the form of *industrial robots* working instead of humans in factories, manufacturing lines, hazardous environment, and even in space. Robots can have *physical parts*, i.e., hardware components, like body, hands, grip, and wheels, and *logical parts*, i.e., software programs that control robots to perform some tasks. However, a robot can also exist purely in the logical form, i.e., as a software, e.g., search engine robots and personal software agents. A robot can be programmed to do certain tasks without interactions with humans, or it can be *interactive*, communicating with human during its operation. Fong et al., in his survey [9], defined a socially interactive robots as a robot for which social human-robot interaction is important. In this work we are concerned with such *interactive robots*.

Research in robotics has made significant progress in the past few decades. Although most researches are concerned with industrial robots, recently more interests has been given to interactive robots. In this chapter we summarize approaches in designing architectures for such interactive robots and present some interactive robot systems that have been developed.

### 2.1.1 Architecture Design Approaches

In this thesis, the term *robot architecture* refers to the internal organization of a robotic system, similar to the definition in [14]. Architectures of robots can be classified in many ways. They can be *biologically inspired*, i.e., it is designed based on the structure of some biological systems, or *functionally designed*, where functionalities of the system are concerned, rather than having similar structures to biological systems. Considering robot internal components, a robot can have a single monolithic component where all sensors and actuators are managed by a single *centralized* element, or it can have a *distributed* or *modular* architecture, where the system is divided into several modules interacting with each other. Each module might have a certain level of autonomy. They can communicate with each other either directly or via a central *blackboard* that facilitates and controls communications among modules. When the robot makes some actions as a reaction to some input events, the process of this event-action behavior might involve the use of some *symbols* to represent such events and actions, or it can process in the signal level, without the use of any symbols.

For robot designers, robot internal design and mechanisms to manage robot behaviors are of interest. To this aspect, researchers often classify robot systems into three types: *deliberative* (systems with intermediate symbolic representation and reasoning, e.g., the sense-plan-act paradigm), *reactive* (layered architecture, with direct connections between sensors and actuators), and *hybrid* (combination of both).

#### Deliberative Architecture

The key of the deliberative architecture is the use of explicit representation in a central component. Symbols are used to represent concepts and objects in the world of interest and in reasoning. Systems using this approach have typical components as shown in Figure 2.1 (partly based on the ideas in [15]). Status or changes in the environment are measured by sensors and transformed into some kinds of symbolic representations. Planner, the central processing element, decides the next action(s) to be done in the form of symbolic operations, and optionally updates the world model, the symbolic model of the world of interest. Action synthesizer translates the symbolic operation into action commands and sends to actuator. Finally, actuators generate physical action(s) to the environment. This model is also called the sense-plan-act model according to the steps of processing.

By using symbols, we can develop complex systems based on these abstracted notions. Human can easily understand what the system is doing and why. And as human's languages also use symbols, the use of them in robot systems might benefit in applications like human-robot dialogue interaction [16]. It has also been shown that learning in robots is much more effective if the robot is operated at the symbolic level [17].

The main drawback of this approach is that the system usually has slow response time because of its sequential processing of information and the need to maintain the world

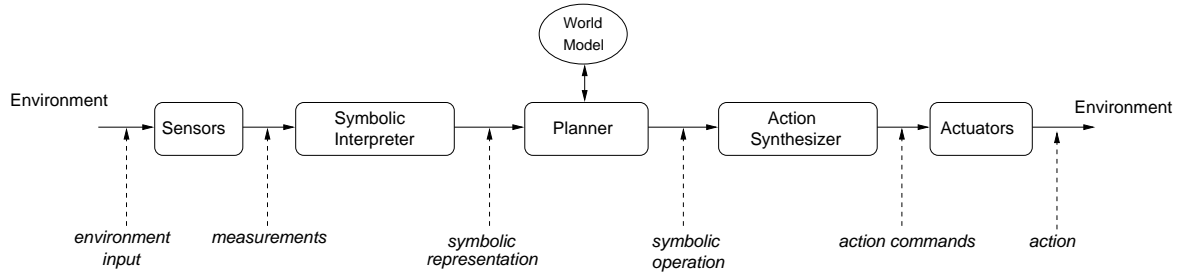


Figure 2.1: Deliberative architecture

model. It is also difficult to make a complete and accurate model of the world which is complex and changing spontaneously. The system is brittle — they can fail in situations only slightly different from those for which they were programmed. Failure in single element can cause a blockade or collapse of the whole system. Also, such problems like symbol grounding and anchoring problem need to be considered [18]. Consistency is required between the real and symbolic world.

A classic example of robot systems that use deliberative architectures is the Shakey robot by Stanford Research Institute (SRI).

### Reactive Architecture

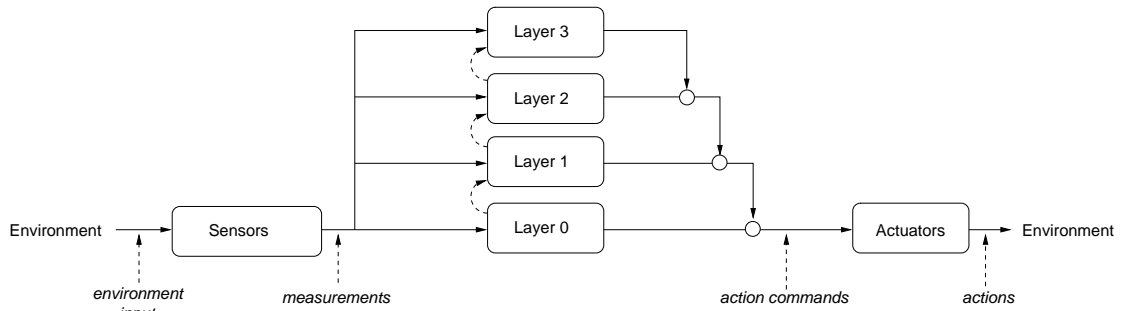


Figure 2.2: Reactive architecture

Because of the problems of deliberative architecture, another group of researchers proposed direct sensory-motor associations without intermediate symbolic representation and reasoning. The most obvious example of this approach is Brooks's layered architecture [19, 20]. In this architecture, the centralized world model is not necessary. The system is decomposed into layers of control as shown in Figure 2.2. Each layer contains a number of finite state machines. It operates asynchronously and gives the system a set of behaviors. Lower level layers do not rely on the existence of higher levels. Layers are combined through



mechanisms of suppression and inhibition, resulting in priority based arbitration [20]. The behavior of the system as a whole is resulted from many interacting simple behaviors.

The major advantage of these systems is that the system is responsive and robust. The overall behavior emerges when it is placed in the environment. No explicit knowledge representation is needed. However, it is very hard to engineer the system to fulfill some specific tasks, and to make it learn from experience [21]. There are also problems of lacking of natural ordering of layers and competence clashes [15].

Since reactive systems are limited by their lack of internal state, the behavior-based approach, as an extension of reactive approach, overcomes this limitation by allowing the underlying unit of presentation, behaviors, to store state [22].

### Hybrid Architecture

Since both deliberative and reactive architectures have their own advantages and disadvantages in certain domains and problems [23], some researchers tried to compromise both approaches and proposed *hybrid* architectures. There are a lot of variations of architectures in this category, depending on the design details, e.g., what techniques are used and how they are combined. An example of a hybrid architecture is SSS (acronym for servo, subsumption, symbolic) architecture for robot navigation [24]. SSS architecture has three layers: Servo, Subsumption, and Symbolic. It tries to combine features of conventional servo-systems and signal processing with multi-agent reactive controllers and state-based symbolic AI systems.

More examples of hybrid architectures are [25], [26], [23], and [27].

### 2.1.2 Distributed Systems

In the previous section, we discussed robot architectures in the control level, addressing the question of how a robot can be designed to act and re-act to the environment through its sensor and actuator modules. Interactive robots are usually integrated systems composed of several hardware and software modules<sup>1</sup>. Based on the architecture, these modules cooperate and form robot behaviors. In this section, we take a brief look at organizations of these modules inside the robots: how a robot system is decomposed into several modules and how these modules interact with each other.

There are several ways to decompose a robot system into multiple modules. Many ideas can be borrowed from the field of Distributed Systems where researchers are focusing on linking several systems together with the purpose to provide global sharing of computational resources, bridge geographic distances, improve performance and availability, allow

---

<sup>1</sup>There are also some robot systems which a robots is regarded as a single inseparable system, i.e., no decomposition into modules. Parts of the system are connected together according to the need of transferring data. However, such systems are rather small. The complexity usually increases rapidly when the size of such systems grow.

distributed problem solving and parallel execution of separate tasks, etc. In a robot system, modules can be sub-systems responsible for complicated tasks with a certain level of autonomy, or simple components that are only responsible for some small tasks or computing resources without any high-level processing. Communications among these modules can be signal-based data transfer or high-level symbolic commands, depending on the designers. Based on the communications among modules, distributed systems can be classified as follows:

- **Distributed Shared Memory (DSM):** A distributed shared memory is a mechanism aimed to allow processes on different modules to access shared data without having to use inter-process communications. Each module is a processing element with its own processor. In fact there are inter-process communications to facilitate this data sharing, but DSM makes these communications transparent to developers. In this scheme, modules are tightly coupled via the shared memory. It is suitable for systems with large data sharing among components. However, adding or removing a module might result in modification of other modules and the whole shared memory structure. Example of distributed shared memory systems are Stanford's SAM [28] and Dosmos [29].
- **Remote Procedure Call (RPC):** Each module has a control over its resource and usually offers services to other modules through a provided interface. This is a mechanism to call remote functions (or procedures) residing on other computing elements instead of within the same program, hence the name "Remote Procedure Call". The toolkit for implementing RPC usually provides abstraction for remote access, making it transparent to the caller. Remote procedure calls can be made directly to the remote module or via a directory server. RPC is suitable for systems where most of communications are service requests among modules and such requests can be fulfilled by using a function-call mechanism. Example of RPC techniques are Sun RPC, Sun's Java RMI, Microsoft DCOM, OMG's CORBA [30], XML-RPC [31], and SOAP [32].
- **Message-passing:** In this scheme, modules are higher-level computing elements with a certain level of autonomy. Each module has a control over its resource and has a goal to accomplish some tasks. They communicate by exchanging messages, usually using high-level symbols. This design is suitable for systems with non-primitive modules cooperating to accomplish some tasks. However, each module must be able to interpret and understand high-level messages exchanging among modules. In this design, modules are often called as agents, indicating some levels of autonomy of them, and systems are called multi-agent systems. Example of message-passing systems are OAA and FIPA [33].

When designing a robot system, there are issues to consider like how should the robot be decomposed into sub-components and how should they communicate. The autonomous-level of modules must also be considered, should they be only primitive and passive computing elements, or higher-level active components with own goals and a certain level of autonomy. In the next section we present some recent interactive robot systems and their internal designs.

### 2.1.3 Interactive Robot Systems

Ishiguro et al. proposed a robot architecture based on *situated modules* and *episode rules* for interaction-oriented robots with large number of behaviors, and a visualizing tool for understanding the developed complex system [25, 34]. The approach is to continue implementing behaviors until humans think the robot has an animated and lifelike existence beyond that of a simple automatic machine. With some influences from behavior-based robot architecture, a behavior-based hybrid architecture is proposed and implemented on a humanoid robot Robovie [35]. The robot interacts with humans based on situated modules. The situated module is an action-reaction pair with pre-condition specifying a robot interactive or reactive behavior. The robot executes one situated module at a time. Episode rules control sequential transitions among situated modules. Priority can be assigned to episode rules. Network of situated modules governed by episode rules forms a complicated switching of robot behaviors. The prototype system has more than 100 behaviors, 102 situated modules and 884 episode rules.

Jijo-2 mobile robot interacts with and learns from humans in the office environment [36, 37]. It features a natural language dialogue system for office services, and can answer queries about people's location. It can conduct route guidance and delivery tasks. The robot accepts human direction commands to reach places and guide people through an office environment. The dialogue manager supports form-based dialog type and maintains the current dialogue state and a list of salient entities referred to by the preceding utterances.

Kawamura et al. proposed a multi-agent robot control architecture called the Intelligent Machine Architecture (IMA) [38]. The system is decomposed into several agents communicating using a distributed software component technology and sharing heterogeneous data structures where information about surrounding entities and actions to be performed are stored. Agents are used to represent hardware, behavior, environment, human, the robot itself, etc.

The Carl interactive service robot can navigate around and make spoken dialogues with human as well as learn new things [39, 40]. It was developed with the aim to integrate communications, action, reasoning, and learning. The multi-process software architecture features a central manager which is an event-driven system using a Prolog inference engine. Human can teach the mobile robot to move around the object.

Minerva tour guide robot [41], the successor of the Rhino museum guide robot [42], is a mobile robot designed to educate and entertain people in public places. The robot navigates in the museum and interacts with people. The robot architecture is distributed and layered.

Etani discussed an architecture for an autonomous, interactive robot based on a multi-agent system [43]. A robot navigation system that can guide people through halls was developed. The system is based on the layered architecture and composed of three layers: Communication, Behavior, and Action. The communication layer maintains the system's internal states, and displays graphical character with voice guidance. The behavior layer manages several environment inputs and executes path planning. The action layer controls the mobile robot, manages input information from location system, and has tasks like collision avoidance.

More reviews of interactive robot systems are available also in Section 2.3.3.

#### **2.1.4 Concerned Difficulties and Issues**

Although robotics and AI technologies have made impressive progress in the recent decades — we have now humanoid robots that can walk, and computer programs that can beat humans in some problems —, the fact that no systems can interact with humans intelligently clearly shows that a lot needs to be done, especially on the integration side. A robot architecture that allows efficient combination of sensory-and-motor devices and their control software, i.e., the robotics part, and the brain of the robot, i.e., the intelligence part, is needed.

Consider the intelligence part, until now there is no such model that provides general intelligence for all robot tasks, instead we have many developed techniques that are good in certain particular tasks. These techniques need to be combined. The question is, what is a suitable general substrate layer that combines various techniques into an integrated robot system. As we cannot expect everything to finish at once, it should be easy to add new techniques and components to the system along the course of research and development.

## **2.2 Frame-based Knowledge Systems**

Frame knowledge representation is one of the primary technologies used for large scale knowledge representation in AI [44]. Frame theory is a theory of representation that stresses a rich symbolic fabric woven out of shared terminals, prerequisite conditions, viewpoint transformations, defaults, expectations, and information retrieval ideas [45]. Frame knowledge technique is used to represent knowledge in many information and experts system. Frame captures the way experts typically think about the knowledge. It is cognitively simple, intuitive and understandable for domain experts. Limited expressiveness of frame-based systems enables tractable inferencing [12].

The theory of frame is introduced by Minsky in [11] in 1974. As it has inspired quite a number of later work in AI, here we quote some selected parts originally explained by Minsky in his paper:

When one encounters a new situation (or makes a substantial change in one's view of a problem), one selects from memory a structure called a *frame*. This is a remembered framework to be adapted to fit reality by changing details as necessary.

A frame is a data-structure for representing a stereotyped situation like being in a certain kind of living room or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed.

We can think of a frame as a network of nodes and relations. The "top levels" of a frame are fixed, and represent things that are always true about the supposed situation. The lower levels have many *terminals* – "slots" that must be filled by specific instances or data. Each terminal can specify conditions its assignments must meet. (The assignments themselves are usually smaller "subframes.") Simple conditions are specified by *markers* that might require a terminal assignment to be a person, an object of sufficient value, or a pointer to a subframe of a certain type. More complex conditions can specify relations among the things assigned to several terminals.

Collections of related frames are linked together into *frame-systems*. The effects of important actions are mirrored by transformations between the frames of a system. These are used to make certain kinds of calculations economical, to represent changes of emphasis and attention, and to account for the effectiveness of "imagery".

The theory, however, provides a rather abstract idea. To use frame in real applications, a more concrete design is required. There are surprisingly large numbers of frame-based systems nowadays. In this chapter we describe a typical frame knowledge model and highlight some interesting frame-based systems that are comparable to our work.

### 2.2.1 Frame Knowledge Model

In the past few decades, AI researchers in knowledge representation have implemented over 50 frame knowledge representation systems [46]. Typical features of the frame-based knowledge model are:

- **Structural features:** Frames are connected together in an IS-A hierarchy. Frames in the lower level down the hierarchy inherit properties from their parents and contain more specific information). Attached to each frame are slots or properties of the frame, usually listed in feature-value pairs with some constraints or facets.
- **Behavioral properties:** Some actions can be triggered when certain events happen. One can attach a procedural script or method to a slot (also called in this case active values), which then will be invoked when the slot value is accessed or modified.
- **Reasoning services:** Frame systems usually provide some sorts of inferencing, e.g., forward and backward chainings [47], and inheritance and slot constraints checking.

In this section, we try to cover some common parts of the frame model which are shared by many systems. We base the explanation on the knowledge model of the *Open Knowledge Base Connectivity* or *OKBC* which aims to give a common standard on the design of knowledge models and access interfaces for frame-based knowledge systems.

OKBC is a protocol for accessing knowledge bases (KBs) in knowledge representation systems (KRSs) [48]. It consists of set of (local or network) operations that provide a generic interface to the underlying KRSs. OKBC has implementations in several computer languages. OKBC specifies elements of the frame-based knowledge model, e.g., constants, frames. Constants are data of basic type integer, floating point, string, symbol, list, and class. A frame is a primitive object that represents an entity in the domain of discourse. A frame that represents a class is called a *class* frame. A frame that represents an individual is called an *individual* frame. A frame has associated with it a set of *own slots*. A slot can have *facets*, which specifies some conditions about the slot. A slot can be provided with a default value. A *class* is a set of entities. Each entity in a class is an *instance* of the class. A class frame has associated with it *template slots* and *template facets* that describe own slot values for each instance of the class represented by the frame. A class can be considered *non-primitive* if its template slot values and facet values can specify a set of necessary and sufficient conditions for being an instance of the class. Otherwise it is considered a *primitive* class, which can have many properties specified but will typically not contain sufficient conditions to be concluded that an entity is an instance of this class. OKBC provides the detailed specification of the knowledge model and a rich set of programming interfaces, e.g., class creation, and slot value retrieval or modification.

Frame shares some similarities to the *Object-Oriented Programming* (OOP) concept. Objects and their properties are similar to frames and slots. However, the purposes of the two models are different: OOP is targeted for programming while Frame is for representing knowledge. The taxonomy and hierarchy concepts of frames are also common in Ontology fields, which has gained much interest recently. In many cases, they are comparable.

### 2.2.2 Frame-based systems

In this section, we review some selected frame-based systems.

Protege-2000 is an OKBC-compatible knowledge-base-editing environment [49, 12]. The goals of this system is to achieve interoperability with other knowledge-representation systems, and an easy-to-use and configurable knowledge acquisition tool. The first goal is accomplished by making the knowledge model compatible to OKBC [48]. Also Protege-2000 extends some features of OKBC in a way that is not prohibited by OKBC, e.g., the use of metaclass. The knowledge model of Protege-2000 is frame-based, with units like classes, slots, facets, axioms, and instances. Classes are concepts in the domain of discourse. Slots are properties or attributes of classes. A slot can be attached to a frame either as a *template slot* or an *own slot*. Facets describes properties including constraints of slots, e.g., cardinality (single, multiple), type (integer, string, instance), and maximum and minimum values. Axioms specify additional constraints. Instances are classes with specific slot values. A frame in Protege-2000 can be an instance of only one class. Every class, slot, facet is a frame. Protege-2000 introduced metaclass, or a template for classes that are its instances. Metaclass describes own slots (and constraints) of a class that instantiates this template. In other words, the template slots defined in the metaclass will become own slots in classes that are instances of this metaclass. The key features of Protege-2000 are (1) the metaclass architecture, (2) the ability to define specialized user-interface components to display and acquire slot values (Protege-200 can automatically generate forms for inputting data), and (3) the component architecture, which allows it to work as an editor for other knowledge-representation systems, e.g., RDF.

FramerD is a distributed object-oriented database designed to support maintenance and sharing of knowledge bases [50]. Unlike typical object-oriented databases, FramerD is optimized for pointer-intensive data structures used by semantic networks, frame systems, and many intelligent agent applications. FramerD is aimed to be robust, high-performance and able to handle large-scale multi-domain contents. Its databases readily include millions of searchable frames and may be distributed over multiple networked machines. The architecture is divided into three layers: Data layer (Dtypes), Object (OIDs) and INDEX (IDX) layers, and Tools (Framer) layer. The lowest data level provides recursively compose-able structures and primitives like vectors, lists, numbers, strings, and also provides object IDs for object references. The object layer provides for the mapping of these references into data layer values. The index layer provides for the maintenance of large inverted indices whose keys and values are arbitrary Dtypes. The top tool layer uses the object and index layers to provide representation facilities via a frame-based “*representation language language*” (RLL). FramerD provides basic operations on units and slots: getting slot value, testing whether the slot includes a value, adding an object to the slot values, and removing an object from the slot values. When the slot is itself an object, the default behaviors are modified by *demons*

associated with the slot description. These demons are expressions in a scripting language stored in symbolic slots specifying what to be executed in case of these actions: get the slot value, test the slot value, add a value, and remove a value.

CODE4 (Conceptually Oriented Design/Description Environment) is a general purpose knowledge management system intended to assist with the common knowledge processing needs of anyone who desires to analyze, debug, and retrieve conceptual in applications, e.g., specification, design and user documentation of computer systems, development of ontologies for natural language understanding [51]. The aim is to develop a tool which helps people to originate, organize, define concepts, understand and communicate ideas at the knowledge level, rather than a system designed to run autonomously. CODE4 is designed to be easily adaptable to many applications, such as natural language processing, software specification and design, expert systems, general terminological analysis, or teaching subjects such as biology or Unix. It features a graphical user interface and claimed to be used by non-computer people in a few days. The knowledge representation used in CODE4 is based on concepts of frame, conceptual graph, object-orientation and description logic. It differs from typical frame-based systems in the generality and uniformity of the knowledge unit in CODE, called *concept*. The role of slots in typical frame-based systems are performed by *properties* in CODE4, which are also concepts themselves. Statement are treated as concepts (with the subject role) that link properties (with the *predicate* role) to concepts.

Soshnikov [52, 53] proposes a JULIA software toolkit for building embedded and distributed knowledge-based systems. The architecture is based on frame knowledge representation and production rules. The approach allows combining in one model static knowledge in the form of slot values, structural knowledge in the form of frame hierarchy, and dynamic knowledge in the form of attached procedures. The main feature of this toolkit is the ability to be distributed over the network and share different knowledge types through the mechanisms of *distributed frame hierarchies* and *remote rules*. Distributed knowledge processing and sharing takes place when different nodes exchange static or dynamic knowledge. The implementation uses Java for representing frames, CORBA for interfacing with other systems, JFMDL (Julia Frame Model Description Language) and XML for knowledge formulation. It also provides an expert system shell in the form of Java applet. Backward chaining inference is supported and a web interface to access the system is planned.

### 2.2.3 Applications of Frame-based systems in Robotics

Since the frame technique is useful for representing the structure of the world, it has been used in quite a few robot systems. Back in 1971, the SHRDLU natural language understanding system [54] also employed this technique. Humans can enter commands in natural language to manipulate the simulated block-world. An example use of Frame-based knowledge technique with a physical robot is the HARIS robot system [55, 13]. The system con-



sists of model-based 3D vision, intelligent scheduler, computerized arm and hand controller, HARIS arm/hand unit and human interface. The system is aimed to help the aged and disabled persons manipulate objects on the desk. The world model is a shared knowledge base working as a communication channel among software modules. It consists of Shape model, which includes shape-related attributes of all potential objects within the scene; Functional model, which represents functional attributes of each potential object and is used in task scheduling and motion control; and Spatial model, which is generated as a result of scene understanding and used for task and motion scheduling and monitoring of robot's motions. Another is the Scheduling model, which contains the knowledge about tasks and motions.

In the HARIS robot system, user can issue commands like "Put a red cup onto a white tray" or "Put a red cup and a black cup onto a white tray". The goal task is added to the task scheduler, which will generate a sequence of primitive tasks to achieve the goal, such as, "reach", "hold", "move", and so on. The motion scheduler generates a sequence of motion primitives to accomplish each primitive task. The command generator receives the motion primitives and executes the motions to the robot arm. Planning of the path of robot arm to avoid obstructing objects on the expected path is done as a part of the motion scheduling using the knowledge stored in the functional and spatial models.

Frame-based knowledge representation is used in the world model. Example of frames that represent a red cup, a cup vessel, and an instance of a cup in the shape model and functional model are shown in the following Table 2.1 and Table 2.2. A cup is composed of a cup vessel and a cup handle. To detect a cup vessel, it requires parallel lines and an ellipse. Based on this knowledge, when a cup is detected, a corresponding class frame is instantiated in the functional model. Description of an example Cup-1 frame is shown in Table 2.3. The simplified image of a cup vessel is shown in Figure 2.3.

<b>Frame: Red-cup:</b>	
<b>A-kind-of</b>	Cup
<b>Color</b>	Red
<b>Has-parts</b>	(Cup_Vessel Cup_Handle)
<b>External_diameter</b>	7
<b>External_height</b>	12
.....	

Table 2.1: Description of a Red-cup frame

Spatial descriptions and relationship among objects as well as tasks are represented similarly using the frame technique. The system is implemented on the generalized frame-based knowledge engineering environment ZERO++ [56].

<b>Frame: Cup_Vessel</b>	
<b>A-kind-of</b>	Vessel
<b>Color</b>	(Red Black Yellow)
<b>Has-parts</b>	(parallel_line ellipse)
.....	

Table 2.2: Description of a Cup\_Vessel frame

<b>Frame: Cup-1</b>	
<b>a-kind-of</b>	Cup
<b>has-parts</b>	(container_1 handle_1)
<b>state</b>	empty
<b>roles</b>	(contain_drink)
<b>color</b>	red
<b>accept_tasks</b>	(reach hold lift carry put release leave)
<b>accept_htypes</b>	(hook grasp nip)
<b>default_htype</b>	hook

Table 2.3: An example of a frame representing a red cup in the functional model

## 2.2.4 Concerned Difficulties and Issues

Frame model has been extensively and successfully used in representing knowledge in many systems. Researchers have improved it in many ways to make it able to represent various types of knowledge in the world of interest. We are concerned with the issues of knowledge updating. If the model is to represent the changing world, first the representation model itself must be able to handle temporal changes and, as a knowledge base, provide accesses to the history data. Second, mechanisms to update the knowledge contents must be provided, i.e., the system must be able to accept new information that represents changes in the world and incorporate them into the knowledge base.

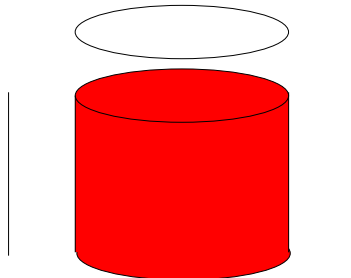


Figure 2.3: Example of a simplified image in identifying a cup vessel using the knowledge in the Cup\_Vessel frame in the Shape model inside the world model

Integrating the knowledge part into a robot system is another issue that has not been much explored. Knowledge base should be there to support intelligent tasks. The problem is how to combine it with other robot components. There are also other techniques that can provide intelligence to the robot; how can they be integrated to the knowledge part and other parts of the robot remains an unsolved issue. It must also be easy and intuitive to development robot applications on the system after the knowledge part has been included.

## 2.3 Dialogue Systems

Development of machines that are able to sustain a conversation with a human being has long been a challenging goal. The need for a dialogue component in a system for human-machine interaction arises from many reasons including that the user does not always express his requirement in a single sentence, rather in a flow of several dialogue turns. Two related research goals adopted by researchers of dialogue [57]. First is the goal of developing a theory of dialogue, generally with the objectives to understand properties of utterances and acts characterizing a dialogue being studied, assumptions about participants' mental states and context for the dialogue to be considered rational, and possible rational and cooperative extensions to the currently observed behavior. The second research goal is to develop algorithms and procedures to support a computer's participation in a cooperative dialogue. Such a human-interacting system in dialogues is called a *dialogue system*. In this work, this second goal is concerned.

Interaction between humans and dialogue systems are usually expected to resemble human-human dialogue, though with less complexities. Levin et al. formalizes a dialogue system as a sequential decision process in terms of its *state space*, set of *possible actions*, and *strategy* [58, 59]. The *state space* is defined by the collection of all variables that characterize the state of the dialogue system at a certain point in time. The *actions* describes what the system can do. For any possible *state*, the *strategy* prescribes the next *action* to perform. As a result of the action and its interaction with the external environment, the system gets some new observations. The new observations are registered and modify the state of the system. This process continues until a final state  $S_f$  is reached. The system starts in the initial state  $S_1$ .  $S_t$  denotes the system state at turn  $t$ . Following this formalization, the process of a dialogue system can be summarized as follows:

$$\begin{aligned}
 &S_t = S_1 \\
 &\text{while}(S_t \neq S_f)\{ \\
 &\quad A_t = \text{NextAction}(S_t) \\
 &\quad \text{invoke}(A_t) \\
 &\quad O_t = \text{environment response to } A_t \\
 &\quad S_{t+1} = \text{NextState}(S_t, A_t, O_t)
 \end{aligned}$$

```

     $t = t + 1$ 
}

```

The field of dialogue systems started to receive more interests in the past few decades. Researchers are considering various features of dialogue systems. For example, man-machine interaction can be system-initiative, where the system directs the conversation; user-initiative, where human initiates the dialogues; or mixed-initiative, a combination of both. The dialogue domains are usually information retrieval and resource reservation, e.g., in systems like Kyoto bus schedule information system [60], MIT Jupiter weather information system [61], and travel planning system [62]. The interaction goal is usually clearly defined, i.e., to obtain desired information or services. Most dialogue systems use speech as a sole communication media. However, multi-modal systems are considered as well, e.g., in SmartKom [63], MATCH [64], CommandTalk [65], and in [66]. There are systems that operate over remote interface like telephone, e.g., Kyoto bus schedule information system [60], [61], as well as embodied systems, e.g., August [67], Jijo-2 [37], Rhino [42], [68], and [40].

Design of a dialogue system is a complex task. Related research areas are speech recognition, speech generation, language understanding, intention recognition, dialogue management, turn-taking, topic management, dialogue context maintenance, etc. For multi-modal systems, topics like gesture-recognition and understanding, synchronization between modes should be considered as well.

In this work, the dialogue management part of the dialogue systems is concerned. Next, we discuss components of dialogue systems and introduce several approaches to dialogue management.

### 2.3.1 Dialogue Systems Components

A traditional components structure of spoken dialogue systems is pipe-line-based as shown in Figure 2.4. Speech input (1) is processed by speech recognition module which gives out a text (2), corresponding to the input speech. The parser and language understanding elements parse and interpret the text, usually resulting in some kinds of communicative acts (3) [69]. The dialogue manager determines these acts and decides the next action to be done in the form of communicative acts (4), which will be transformed to text to be spoken out (5) by the text generation module. Finally, the text-to-speech module generates a speech output (6) from the text in (5). Components like speech recognition, understanding, text-to-speech, and text generation modules might optionally consult the dialogue manager during their operations (7), (8), (9), (10) to ask about conversation history, current context, and other information that is useful to them.

In short, tasks of the dialogue manager can be reduced to two basic actions [70]. First, it interprets observations (usually user input) in context, and updates the internal representation of the dialogue state. And second, it determine the next action of the dialogue system

according to some dialogue policy. The dialogue manager is also expected to take care of identifying and recovering from possible communications errors, e.g., from speech recognition and language understanding [71].

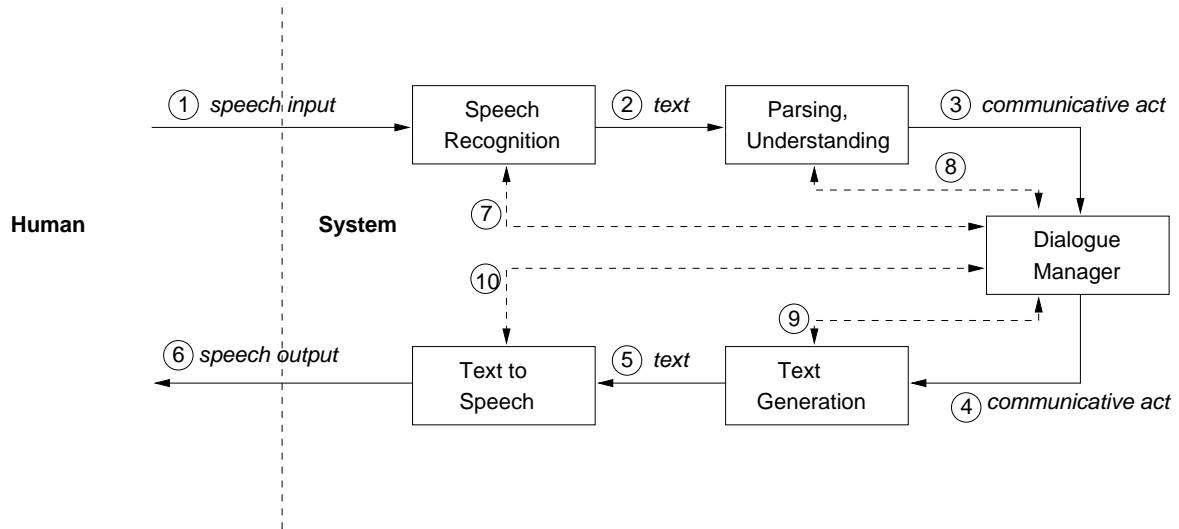


Figure 2.4: Classical pipe-line structure of a spoken dialogue system

Multimodal dialogue systems handle various modes of input and output apart from speech, e.g., visual input via image processing element, input tactile sensors, and robot poses as output. The structure in Figure 2.4 can be generalized to support multi-modal input and outputs. This is shown in Figure 2.5. Instead of sequential processing, the central component dialogue manager might also have a blackboard architecture where other systems components can communicate with each other via the central blackboard.

While these steps are common to almost all the dialogue manager, each step is not trivial and hence has led to a proliferation of different approaches [70]. Next we discuss approaches available to date.

### 2.3.2 Dialogue Management Approaches

Several surveys of dialogue systems report different categorization schemes [72]. Cohen defines three approaches to modeling dialogue: dialogue grammars, plan-based models, and joint action theories of dialogue [57]. However, Xu treats joint action theories of dialogue as a further development of the original plan-based approach, and presents only two groups of approaches: pattern-based (dialogue grammars) and plan-based approaches [73]. McTear classifies dialogue systems into finite-state-based, frame-based, and agent-based systems [74]. Allen grouped dialogue systems into systems that use schema or frames, planning, models of rational interaction, and finite-state models [75]. Robinson stated in [70] that apart

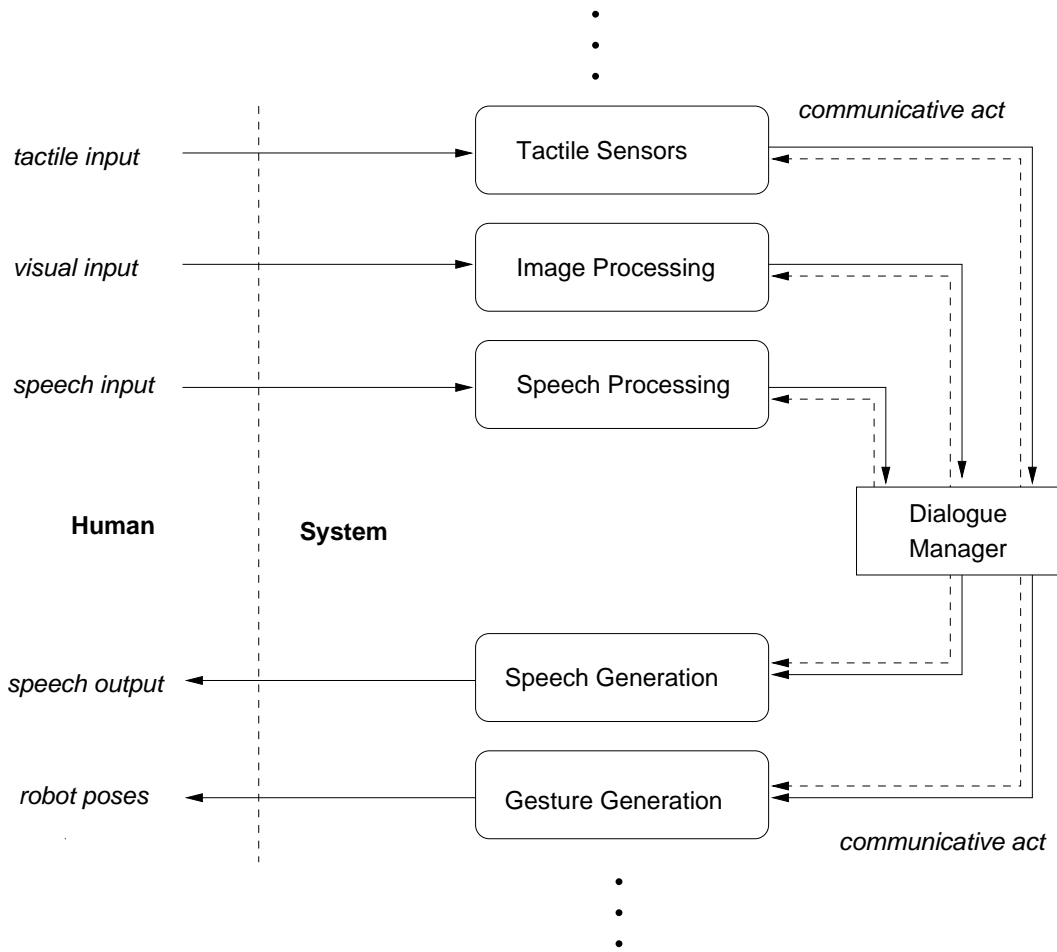


Figure 2.5: Multimodal dialogue systems have more modes of input/output,

from the grammar-based approaches, more sophisticated approaches to dialogue management can be characterized by a separation between the data structure that stores the current state of the dialogue, and the specification of the dialogue policy. There seems to be no consensus on how to classify dialogue systems. However, everybody seems to agree on the two common grammar-based models: finite-state-based, and form-based (also known as form-filling and frame-based) models. The rests are usually considered “more sophisticated” models, e.g., plan-based and information-state approaches.

In this section we explain these approaches, followed by selected examples of developed systems.

### Finite-State Model

In this model, a dialogue is represented by a finite state network. Each node in the network represents a state of the interaction. Attached to each node is a dialogue prompt and gram-

mar at that point of conversation. User input results in transition to a different node, which has a link from the previous node. The dialogue has an explicit flow governed by this transition network, i.e., the dialogue policy is embedded in the network topology. This model is simple and suitable for simple system-initiated dialogues. The fact that every possible states and transitioned must be prepared beforehand makes it difficult to handle complex dialogues where there are a large number of states and transitions.

Examples of systems that use this model are MATCH [64], CommandTalk [65], and MAILIN [66]. There are also toolkits for dialogue system development using this approach, e.g. CSLU [76] and SpeechBuilder [77].

### **Form-based Model**

Although finite-state-based systems have advantage that they are relatively straightforward to design and their behavior is predictable, the disadvantages are that the dialogue tends to be inflexible and machine-initiated [78]. User cannot deviate from the system's plan. Form-based systems (also known as frame-based- and form-based-systems) is based on the policy that the system tries to fill an electronic form by prompting the user for unfilled slots [78]. Each slot in the form has an associate prompt that guides the user through the dialogue. However, the user is not restricted to always follow the system's lead and can take the initiative in the dialogue, e.g., he or she can provide the system with the information that has not yet been requested. Once the form is filled, i.e., all the required input information is received, an action to do something, e.g., reserve a bus seat, is fired.

Form-based model allows mixed-initiative interactions and more flexible dialogues. Example of systems that use this model are the August spoken dialogue system [67], the CMU Communicator [79], and the STAR dialogue manager [80]. Forms are also used as the key component in the Voice Extensible Markup Language (VoiceXML) documents [81, 82].

### **Plan-based Model**

Although the form-based model allows more flexibility than the finite-state one, it is still limited in case that the next system action depends much on the system context or state, which may change dynamically during the course of dialogue, and is difficult to be determined beforehand, for example, in case of negotiation dialogue in the tasks that involve planning and collaborative interaction. Plan-based models of dialogue [83, 84] are based on the observation that utterances are not simply strings of words, but rather the observable performance of communicative actions or *speech acts* [69], such as requesting, informing, warning, suggesting, and confirming [85]. Humans do not perform things randomly, but rather they plan their actions to achieve various goals. Hence an utterance is not to be interpreted only literally but also the intention must be considered. A key element of this approach is the modeling of utterances as speech acts, consisting of roles, preconditions, constraints, and effects

[74]. A plan to achieve a goal typically involves chaining together a series of speech acts (of both parties) in the correct sequence. The grammars-based dialogue model describes what happens in dialogues at the speech act level and cares little about why. While plan-based model tries to explain why agents act in dialogues, but at the expense of complex representation and reasoning. In other words, the first is shallow and descriptive and the latter is deep and explanatory [73].

The problems of planning approach are that it is difficult to recognize of the utterance's communicative, and wrong recognition can lead to incorrect interpretation of speaker's plan; the processes of plan recognition and planning, which involves chaining from preconditions of plans to actions, can in complex cases become combinatorially intractable [74]. Examples of plan-based dialogue systems are [86], [87].

### **Information State Approach**

Considering that the grammar-based approach is suitable only for simple and structured dialogues, and the plan-based approach has drawbacks as being opaque, given the large amount of procedural processing and lack of a well-founded semantics for plan-related operations, the information-state approach [88, 89] is intended to combine the strengths of each paradigm. The information state may include aspects of dialogue state as well other things like obligation, commitment, beliefs, intentions, plans, etc. The key to this approach is identifying relevant aspects of information in dialogue: how they are updated and how updating processes are controlled [89]. In grammar-based models, the dialogue proceeds, based on the result in the current state, to another pre-defined allowable states. Instead, in information state approach, the next move relies not only on the current state but other information available as well. It is characterized by the following components: a specification of the contents of the information state of the dialogue, the data-types used to structure the information state, a set of update rules covering the dynamic changes of the information state, and a control strategy for information state updates [90].

The first implementation of the information-state approach was the Prolog-based TrindiKit [89]. Derivative work in this approach are for example DIPPER [90] and WITAS [91].

Table 2.4 summarizes advantages and disadvantages of the previously mentioned dialogue management approaches.



Dialogue Management Approaches	Advantages	Disadvantages
State-based	<ul style="list-style-type: none"> <li>• simple and robust</li> <li>• user' responses are restricted, easy for speech recognition and understanding</li> <li>• a lot of toolkits available, e.g. CSLU [76], SpeechBuilder [77]</li> </ul>	<ul style="list-style-type: none"> <li>• inflexible and not natural dialogue flows</li> <li>• only supports system-initiative dialogues</li> </ul>
Form-based	<ul style="list-style-type: none"> <li>• more flexible than state-based, support mixed-initiative dialogues</li> <li>• a lot of toolkits available, e.g. CMU Communicator [79]</li> </ul>	<ul style="list-style-type: none"> <li>• inefficient for less structured dialogues where the next system's action is difficult to be determine beforehand</li> </ul>
Plan-based	<ul style="list-style-type: none"> <li>• suitable for dialogues involving planning or negotiating to achieve some task, where the next dialogue depends much on the context and system state</li> </ul>	<ul style="list-style-type: none"> <li>• difficult to infer communicative acts and speaker's plan</li> <li>• process of plan recognition and planning is computing-intensive and can be intractable</li> </ul>
Information State	<ul style="list-style-type: none"> <li>• Combination of both grammar-based and plan-based approaches, using aspects of dialogue states as well as allowing detailed semantic representations and notions of obligation, commitment, beliefs and plans. [90]</li> </ul>	<ul style="list-style-type: none"> <li>• High complexity (the TrindiKit implementation), and testing and debugging is difficult [90]</li> </ul>

Table 2.4: Comparison of several dialogue management approaches

### Other approaches

To avoid the use of a general algorithm to define the dialogue policy, Robinson et al. stated in [70] that more sophisticated approaches to dialogue management attempted to separate between the data structure that stores the current state of the dialogue, and the specification of the dialogue policy. A dialogue model based on frames, transition networks and list structures is proposed. The dialogue states are represented using frames. However, these frames do not imply anything about dialogue policy, instead dialogue policies are defined separately for each task frame, thus gives the designer more freedom and flexibility.

The Circuit-Fit-It Shop system [92] helps users fix an electronic circuit by engaging in a spoken dialogue with the user. Dialogues are used in the acquisition of axioms that are missing but required to complete a task step. The next dialogue is determined dynamically based on the current situation and the user's current state of knowledge. Theorem prover is used to determine task completion.

Most dialogue systems are function-based, composed of series of inter-processing units interfacing each other via representations. However Lemon et al. stressed the importance of interaction level phenomena and introduced parallelism by proposing multi-layered dialogue architecture separating the content and interaction layers [93].

### 2.3.3 Dialogue Systems

Dialog systems have been developed by many groups and are now in use in many applications. Here we some systems that have interesting features related to our work.

SmartKom is a virtual communication assistant, visualized as a life-like character on a graphical display. It features a multimodal dialogue system that supports spoken dialogue, graphical interfaces and gestural interaction modes [63]. The aim is to propose new computational methods for the seamless integration and mutual disambiguation of multimodal input and output on a semantic and pragmatic level. It supports situated understanding of possibly imprecise, ambiguous, or partial multimodal input, and the generation of coordinated, cohesive and coherent multimodal presentations. The underlying integration software is based on the previous work in the Verbmobil's testbed [94]. The dialogue manager has a multi-blackboard architecture with parallel processing threads that support media fusion and media design processes. Multiple modules in the system communicate via multiple blackboards. Interaction management includes representing, reasoning, exploiting models of the user, domain, task, context, and the media. An XML-based markup language M3L modality-free semantic representation is used for representing multimodal content.

The FaSiL Project proposed a practical conversational dialogue management approach with a list-like structure [70]. It supports mixed-initiative dialogues, i.e., the user can barge-in at anytime, while the system-driven backbone is provided to guide new users. Components are highly modularized and independently specified. The User Intention Set (UIS) is a

list structure used to manage commands that are currently being processed. Dialogue states are represented using frames: Task frames and Command frames. In contrast to form-based approach, these frames do not imply anything about the dialogue policy. Separate dialogue policies are specified for each task and command frames.

The Queen's Communicator project investigates the use of the Object Oriented (OO) technique in spoken dialogue management [95]. Generic and domain-specific dialogue behaviors are separated. Dialogue components are modeled using objects, e.g., dialog manager, discourse manager, enquiry expert, dialog frame, discourse history, and dialog server, and set of rules, e.g., heuristic rules, user-focused rules, and database-focused rules. It uses the DARPA Communicator architecture and the Galaxy hub [96, 97].

The August project delivered a spoken dialogue system with a goal to collect spontaneous speech data from people with no experience with the speech technology or computers by putting the system as a kiosk in public place. [67]. It features a simple dialog manager using form-based approach. The form has feature/value pairs with pre-defined answers.

Allen [87] discusses an architecture for a generic dialogue shell with the focus on practical dialogues, dialogues with an aim to accomplish some specific tasks. A new architecture for TRIPS (The Rochester Interactive Planning System) is designed to support plan-based tasks in multiple domain, as an improvement from the previous train route finder TRAINS system. The new architecture follows the plan- and agent-based approach and has six modules: discourse context management, reference resolution, intention recognition, behavioral agent, plan manager and response planning. All modules have accesses to a common semantic hierarchy and a world knowledge manager containing domain specific reasoners and knowledge bases. The behavioral agent, upon receiving a notification stating that the user has initiated a problem solving act, can choose to find out the answer, ask for clarification, notify a failure, or ignore the question (if other more important information is waiting). Basic dialogue system components, e.g., the plan manager and behavioral agent, are separated from more domain-specific components. Components communicate using a KQML-based message-passing communication scheme rather than RPC or OO method calls.

Galaxy-II [97] is an architecture for conversational system development and a successor to the Galaxy architecture of the DARPA Communicator project [96]. It supports development of client-server systems for accessing on-line information using spoken dialogue. It has been used in many application domains, e.g., restaurant guide and weather information. The architecture is client-server-based with a central hub facilitating the communications among components (also called *servers*). There are servers for speech recognition, language understanding, language generation, speech synthesis, etc. Hub-server interaction is controlled by a script. A script includes a list of servers (host, port, and operations that the server supports) and a set of programs. Each program is a set of rules, where each rule specifies an operation, a set of conditions in which the rule will fire, and a list of input and output

variables for the rule as well as other variables. When a rule fires, the input variables are packaged into a token and sent to the server that handles the operation. The hub expects the server to return token containing the output variables at a later time. Semantic frames which are named and typed data structure with fields like clause (high-level request act, e.g., display, record, repeat, reserve, topic (noun phrase), and predicate (attributes), are used as representations shared between servers.

RavenClaw features dialogue management using a hierarchical task decomposition and an expectation Agenda [98]. It is a successor to the Agenda architecture used in the CMU communicator [79]. It is a two-tier architecture with a clear separation between domain-specific dialog task specification and domain-independent discourse behavior specification. Hence it is rather domain-independent and at the time of writing has applications in five domains, e.g., F/A 18 aircraft maintenance tasks, Pittsburgh bus schedules information, and conference room reservation. A dialogue task is represented using a hierarchical tree with a notion of context (parent-child) and a default ordering of actions (left-to-right traversal or more sophisticated policy). Leaf-nodes in the tree are fundamental dialog agents. There are four types of fundamental agents: *Inform*, *Request*, *Expect*, and *DomainOperation*. Each agent has execute routine and holds a set of pre-conditions and triggers, and a completion criterion. Non-leaf nodes are called dialog agencies, for example *Login* and *GetQuery*. They control the execution of the subsumed agents and capture the higher level temporal and logical structure of the dialogue task. RavenClaw provides a rich set of conversational strategies in the form of dialog agencies. These include grounding behaviors (e.g. confirmations, disambiguations, and channel re-establishment), turn-taking and timing behaviors, as well as other generic dialog mechanisms like the ability to handle requests for help, repeat the last utterance, suspend and resume the dialog, start over, and re-establish the context [98]. The dialog engine has a task execution stack which captures temporal and hierarchical structure of the current dialog. Topic switching is possible, e.g., by saying “suspend”. Expectation agenda is a data structure describing what the system expects to hear (top down traversal) useful in reference resolution.

JASPIS [99] is an adaptive speech application architecture for synchronized distributed spoken dialogs. It is based on agents and provides mechanisms for distribution, coordination, dynamic selection of agents, and a common structure for shared information management. The architecture introduces the *agents*, *managers*, and *evaluators* paradigm. Agents are atomic, stateless, and compact, hence support highly modular systems and reusability. Managers coordinate agents. Each manager uses an evaluator to select which agent is the most suitable in each situation. JASPIS has two logical levels: core infrastructure and collection skeletons (modules). Core infrastructure is much similar to the Galaxy-II [97] and OAA [100]. The architecture uses a combination of the blackboard-based and message-based techniques. Agents do not communicate directly but use shared hierarchical blackboard-type

information storage for indirect communication. How the information is actually stored in this storage is not defined, but accesses to it are governed by the *Information Access Protocol*, which is an XML DTD. Interaction manager, like Hub in the Galaxy-II architecture and Facilitator in TRIPS, is more like a coordinator than controller. Interaction manager gives local managers possibilities to act based on triggering events received from the Information Manager. XML-RPC is used in infrastructure level communication. Annotation graph is used for storing and exchanging linguistics information.

Lemon et al. discusses a dialogue system for WITAS UAV (Unmanned Aerial Vehicle), a small robotic helicopter with on-board planning and deliberative systems and vision capabilities [91, 101]. The robot can carry out activities like flying to a location, following a vehicle, and landing. Human operators can specify mission goals and these activities during the dialogue interaction. Emphasis is placed on multi-tasking and interleaved dialogues, and collaborative activities. The dialogue system is based on the information state and dialogue move concept. System components, namely, natural language parser and generator (NL), speech recognizer (SR), text-to-speech (TTS), interactive display with deictic reference support (GUI), dialogue manager (DM), and Activity Layer for robot control, are designed as agents connected to each other using OAA (Open Agent Architecture) [100]. Activity model, used by the Activity Layer, contains the knowledge about how higher-level activities can be decomposed into sequences of atomic actions (which the robot innately understands) with pre- and post-conditions. For example, a LOCATE activity can be decomposed into three steps of WATCH-FOR (looking for an object), FOLLOW-OJB (follow it), and ASK-COMplete (ask the user if the mission is complete). The Conversational Intelligence Architecture (CIA) defines the components inside the dialogue manager. The Dialogue Move Tree (DMT) is a tree-based data structure used as a message board to keep a record of the utterances made by both the system and user. Each sub-tree of the root node represents a thread in the conversation and each node represents an utterance. The Active Node List (ANL) keeps track of which nodes in the DMT tree are active, sorted by their relevances to the current discourse. The Activity Tree (AT) represents the current, past, and planned activities of the system such as moving to a location. Each node in the tree describes an activity in terms of slots and values, similar to in the frame model. The System Agenda collects all of the utterances that the system intends to produce, which will be synthesized (and optionally aggregated) by the generation module. The Pending List collects questions that the system has asked but the user has not yet answered. The Saliency List keeps track of all the noun-phrases used in the conversation so far, ordered primarily by recency for anaphora and deictic resolutions. Lastly the Modality Buffer stores graphical display gestures by the user, for resolution of deictic expressions if necessary. Incoming utterance from user is translated to a logic form representing its syntax and semantics. Each node in the ANL is checked if it matches the utterance. When the match is found, a new node representing the new di-

dialogue move is created as a child of the matched node, because the new utterance from the user was relevant to this particular conversational thread, i.e., this sub-tree in DMT. Lemon also explored further in the area of context-sensitive speech recognition and interpretation of corrective fragments [91].

Hygeiorobot is a mobile robotic assistant for hospitals [102]. It features a spoken dialogue system intended for people with little or no computing experience. The robot performs simple tasks in hospitals like delivery of messages and medicines to particular rooms and interacting with hospital staff via spoken dialogues. The dialogue system architecture is similar to that in Figure 2.4. The system adopts the state-based dialogue management approach and based the development on the CSLU Toolkit [76]

### 2.3.4 Frame-based Knowledge Technique and Dialogue Systems

Uses of knowledge back-end in dialogue systems are prevalent in dialogue systems. In the Prolog-based Sundial [103] system, multiple languages and tasks are supported by the use of multiple static knowledge bases. There are also uses of frame-like hierarchical knowledge technique. A common one is to store the world or domain knowledge, e.g., the common semantic hierarchy in TRIPS [87]. In the form-based approach of dialogue management, frames with slots are used as dialogue forms. The general dialogue policy is to try to fill these slots; once all slots are filled, an action like database query can be started.

In Vox's FASiL system, separation of data structure that stores dialogue states from dialogue policies is proposed [70]. Frames are used to store dialogue states, and state transition network is used to specify dialogue policy. The Jaspis architecture is designed to support distributed spoken dialogs using multi-agent techniques [99]. Shared system knowledge is organized hierarchically and made accessible via an access protocol defined using XML-DTDs.

Other use of frames is, for example, in the Galaxy-II dialogue architecture, semantic frame representation is used for inter-server communications [97]. RavenClaw dialogue manager in the CMU Communicator project has another use of tree-like structure to store dialogue tasks to be executed [98]. Similar to Frame model, object oriented technique is used in the Queen's communicator [95]. Things like dialogue frames, domain experts are modeled as objects. Set of user- and database-related rules are used to manage system behaviors.

### 2.3.5 Concerned Difficulties and Issues

We are concerned with the issue of integration of the dialogue management function into a knowledge-based robot system. While there are many uses of knowledge techniques in dialogue systems, most of them are to employ a task-dependent knowledge base to store some dialogue-related data. This made it difficult to share the knowledge with other robot applications (which might have another sets of knowledge bases) running on the same robot

system. If we are to integrate many robot applications in a single platform, we need a general shared knowledge layer that will be the playground for all applications including the dialogue management and enable efficient knowledge sharing among them. This knowledge layer, on the other hand, must provide necessary facilities needed in managing dialogues.

## Chapter 3

# Knowledge-based Distributed Robot Architecture

This chapter introduces a distributed architecture for robots where several robotic hardware and software components are designed as agents and connected on the network.

### 3.1 Introduction

Research on humanoid robots has progressed rapidly during the last decade. The robots can now even imitate human's biped movement. Also we observed major progresses in the research fields of Artificial Intelligence (AI) and man-machine interfaces, e.g., advancement in face recognition, voice recognition, voice synthesis, and natural language processing. However, most robots developed still lack the ability to interact with humans in a natural way. Robots can barely interact with humans autonomously and intelligently. One reason for this might be that researchers tended to focus on various different components of intelligent behavior (e.g., reasoning, learning, and problem solving) in isolation [21]. Integration of these components, hardware and software, to achieve an intelligent integrated robot is a challenging but often neglected topic. In this work, we proposed a platform-based robot architecture which allows integration of robotic devices and intelligent software components with various functional modules, and at the same time provides a mechanism for managing robot behaviors as a whole based on these combined elements.

Various robot architectures have been proposed. Prevalent in traditional AI researches are symbol-based systems which make use of symbols to representation things in the world of interest and in reasoning. In contrast, reactive systems consider direct sensory-motor associations without intermediate symbolic representation and reasoning. Both symbolic and non-symbolic approaches have advantages and disadvantages in certain domains and problems. Some researchers proposed hybrid architectures mixing both approaches. Although



these systems are effective in individual applications they are targeted for (e.g., dialogue management, behaviors switching, path learning from human instruction), it is difficult to add new applications to the system and share the knowledge among applications. The underlying architecture is usually influenced by target applications and system components. In case that the target goal is changed, which might require new applications and new components in the system, the architecture needs to be adjusted. When an architecture is very application-oriented, developers have to learn specific features found only in that system. Hence, designing an application is more complicated, compared to the systems with application-independent general design.

We propose a robot architecture based on a general knowledge platform serving as common ground for various robot applications. The architecture is distributed. Various robotic hardware and software components are designed as agents and connected on the network. This allows easy modification of system parts. The architecture is hybrid: symbol-based at the center and behavior-based in the agent-level. Agents can have their own low-level behaviors and can communicate with each others directly. The special agent *knowledge manager* is a central module for higher-level task planning and execution.

This chapter discusses an overview of the architecture and roles of the knowledge manager, the symbolic part. The frame knowledge model is used as the presentation in the knowledge manager.

## 3.2 System Architecture

In this section, an overview of the proposed robot architecture is given. A robot is composed of various software and hardware components. These components usually have different usages and programming interfaces depending on their types, programming languages, computing platform and manufacturers. Since all these components are to be combined into one integrated system, the robot architecture should allow efficient cooperations among them. And to cope with rapid changes of technology, developments in different parts should be independent, i.e., they can proceed without having to wait for other parts.

### 3.2.1 Primitive Agent

To accommodate these needs, in our architecture, a robot is divided into small networked components called *primitive agents*<sup>1</sup>. Each primitive agent is either responsible for a specific task, e.g., speech recognition, face detection; or representing a certain robotic device it is connected to, e.g., a video camera and speakers. Primitive agents offer services to other agents on the network through its *interface*. Technically, a primitive agent is a software performing a

---

<sup>1</sup>The term *agent* here means an individual computing element that performs a specific task using the resource it controls. It can be autonomous or non-autonomous, intelligent or not intelligent.

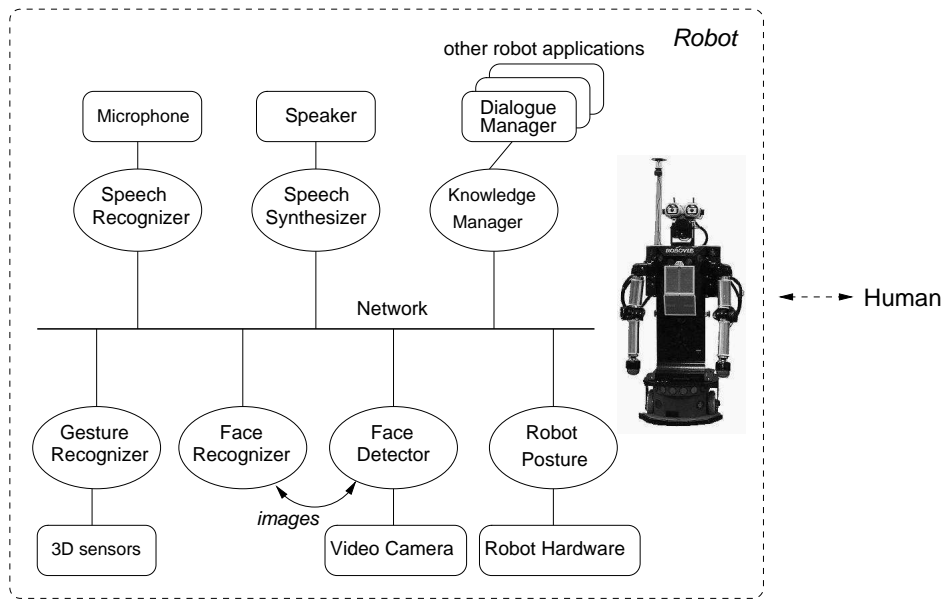


Figure 3.1: Robot as a network of primitive agents: speech recognizer, speech synthesizer, knowledge manager, gesture recognizer, face recognizer, face detector, and robot posture

certain task wrapped by a network server. The server waits for procedure call requests from other agents. When a request arrives, it accepts and forwards to the appropriate code that performs the real processing. Figure 3.1 illustrates our prototype robot system as a network of seven primitive agents.

This architecture allows integration of existing and future components easily. Communication between agents should be in a standard protocol which is independent from the agent's computing platform and programming language. The system is scalable as workload can be distributed to many machines.

As an example, a part of the interface of a face detector primitive agent is as follow:

- *string getStatus()*: check agent's status
- *string ping()*: check agent's reachability
- *void setImage(base64 imagecontents)*: set the input image contents
- *string getFaceLocations()*: do the face detection, and return face location(s)

Other primitive agents can make use of the face detector agent by calling the provided procedures, e.g., *setImage*, *getFaceLocations* to find faces in the image.

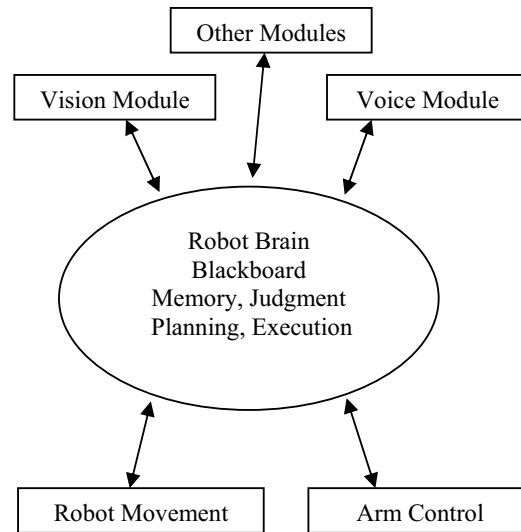


Figure 3.2: Platform approach for robot architecture

### 3.3 Agents Collaboration

Agents are classified as sensor agents, actuator agents, and a special agent *knowledge manager* (KM), which acts as the brain of the system, handling higher-level tasks. Other agents are primitive and passive. KM functions as a blackboard, knowledge processing brain, memory, and does the judgement, task planning and execution. The architecture is considered a *shared-knowledge* multi-agent system, since the knowledge of the whole system is stored in the KM and shared by all agents. The topology of our platform-based robot architecture is shown in Figure 3.2.

However, a low-level agent can also handle some tasks by itself. For example, the face detector agent continuously processes images received from video cameras. When a face is found, it sends a message to the knowledge manager. At the same time, if the detected face is not in the middle of the image, the agent directly contacts the robot posture agent to move the robot's neck so that it follows the human face. Agents can communicate directly, e.g., the face detector agent transfers detected face images to the face recognizer agent.

Agents collaborations are illustrated in Figure 3.3. The big rectangle represents a robot. Inside the robot there are a knowledge manager agent, which is considered in the knowledge (or symbolic) layer, and other agents, which are considered in the sensor (or non-symbolic) layer. In case (1), an event is received by the agent A, and causes it to response by generating an action back. This event-action mechanism is managed solely by the agent A. This can be used in the behaviors that only one agent is involved and it can make a decision immediately, e.g., safety protection behaviors. For example, a robot wheel should not try

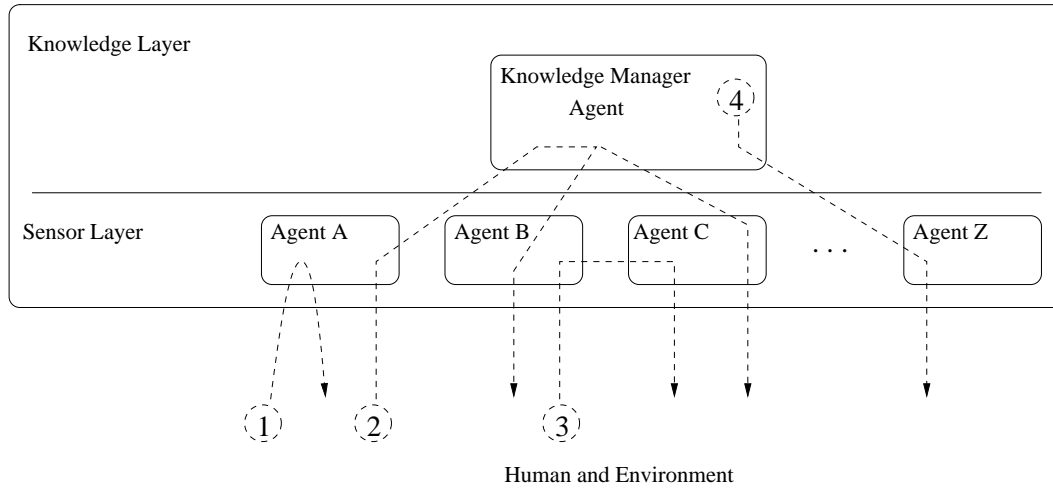


Figure 3.3: Four types of collaboration among agents that compose a robot (the big rectangle).

to move further if it is obstructed by some objects, otherwise it might cause damages to its own hardware. In case (2), the sensor agent A receives an event but can not make actions by itself. It sends the information to the knowledge manager for higher knowledge-level processing. The knowledge manager generates actions to be carried out by agent(s), e.g., to the agent B and C. An agent can also contact other non-knowledge-manager agents directly without having to go through the knowledge manager. This is illustrated in the case (3) in the diagram, and is similar to the previous example of a face detector agent. In case (4), the knowledge manager can generate actions to be done by an agent Z without any events from agents, but instead from other internal triggering events, e.g., clock tick.

The next section discusses the conceptual design of the knowledge manager.

### 3.4 Knowledge Manager Agent

The knowledge manager agent is designed to have three primary responsibilities. First, with the information from sensory agents, KM makes sure that the representation of the world of interest, i.e., the World model in its knowledge base, is up-to-date. As shown in Figure 3.4, when things changed in the real world and those changes are captured by sensor agents, KM will update its World model to reflect those changes. Second, from changes in the knowledge contents and/or passing time, KM generates proper actions as output via actuator agents based on the knowledge in the Tasks/Actions model. This task can be also shared by external knowledge manipulators in case that the action management mechanisms provided in KM do not suffice. Lastly, as a knowledge base, KM answers queries from other agents using the knowledge it has.

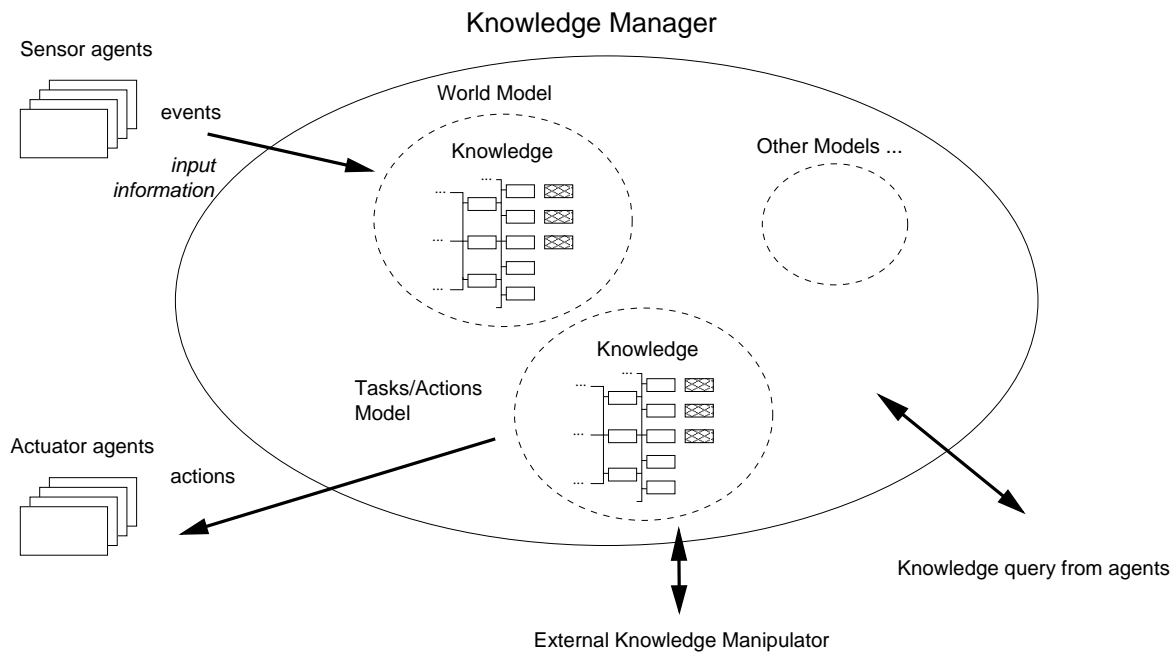


Figure 3.4: Knowledge manager's knowledge model and interactions with other components

Based on the above concepts, robot actions are generated by the KM in four ways:

1. **Event  $\rightarrow$  Action:** Incoming event(s) from sensor agents directly causes output action(s)
2. **Event + Current state  $\rightarrow$  Action + New state:** In a certain system state, incoming event(s) from sensor agents causes updates to the existing knowledge, e.g., the World model, changing the system into a new state and consequently generating output actions.
3. **Passing time  $\rightarrow$  Action:** With no incoming event form outside, the passing time triggers KM to generate output actions.
4. **External manipulator  $\rightarrow$  Action:** Actions are generated by external knowledge manipulators.

### 3.5 Technical Design

The design goal is to have an architecture in which various components or agents cooperate with each other effectively. The architecture is chosen to be distributed because these components might reside on different machines. Interaction between components developed

in different programming languages must be supported. Also the use of the architecture should be intuitive and simple.

### 3.5.1 Communications among Primitive Agents

Software in each primitive agent has different programming interfaces depending on their languages, usages, and manufacturers. Some runs only on some specific platforms. In our demonstration system, for example, the knowledge server primitive agent is written in Java and hence multi-platform, while the face recognizer software is written in C and developed on UNIX machines. Since all these primitive agents are to be integrated, they must be able to communicate with each other effectively. Therefore an effective communications mechanism in this heterogeneous and distributed system is needed.

There are a large number of research and development in the area of agents communications and agents software platform [104], ranging from the distributed software frameworks like CORBA, DCOM, to the more comprehensive agent platforms like FIPA-OS, ZEUS. FIPA [33] aims to create standards among agent platforms. In the DARPA Communicator project, the Galaxy Architecture [96] was used in the spoken-dialogue system where various kinds of components are communicating.

We evaluated various technologies and finally selected XML-RPC [31] as the communication protocol between primitive agents. XML-RPC enables remote-procedure-call (RPC) across various computing platforms and programming languages. It is simple and light-weight. XML-RPC messages are transported in the text-based XML format which is open, standardized, and easy for human inspection. There are many XML-RPC implementations in various computer languages, e.g., C/C++, Java, Perl, and Python, and for various operating systems, e.g., GNU/Linux, Microsoft Windows, and Sun Solaris. We consider the simplicity and the cross-platform features of XML-RPC as its main advantages. XML-RPC is very light-weight in term of resource consumption and is less complex, compared to other techniques like CORBA and SOAP, and it provides the necessary functions [105].

On the other hand, by using XML as data format, the amount of data to be transfered is much larger than that of binary protocols. Using gzip extension of HTTP-1.1 might alleviate the problem but more computing power is needed to compress and decompress the data. Moreover, as XML is a text-based protocol, all data are sent unencrypted over the network. More security, if needed, can be achieved by the use of HTTPS at the price of computing resource for encryption and decryption. However, these issues are not critical in our system.

An example of an XML-RPC message representing a remote-procedure-call to the remote function *ping(int)* passing a parameter of type integer with the value 1 is shown as follow:

A primitive agent is therefore basically a piece of software performing a specific task (e.g., speech recognizer) or representing the hardware (e.g., robot's neck) wrapped by an XML-RPC server. This server waits for requests from other agents. When a request arrives,

```

<?xml version="1.0"?>
<methodCall>
  <methodName>ping</methodName>
  <params>
    <param>
      <value><i4>1</i4></value>
    </param>
  </params>
</methodcall>

```

Table 3.1: Example of an XML-RPC request to a remote function *ping* with one integer parameter

it accepts the request and pass it to the appropriate part of code that performs the real processing.

### 3.5.2 Primitive Agent Abstraction

Although primitive agents can be very different, they share some common properties. For example, all primitive agents should provide a mechanisms for other agents to check their status and test the reachability. Therefore we designed the *generic interface* for such common tasks. Two generic functions *getStatus()* and *ping()* are for checking agent's status and testing the reachability respectively. The verbosity of the status messages can be set by function *setDebug()*. An example generic interface (in pseudo code) for all primitive agents is as follow:

- *string getStatus()*
- *string ping(integer i)*
- *void setDebug(boolean setDebug)*

Moreover, similar primitive agents might also share some properties, e.g., two face detector agents using different face detection algorithms would have the same interface that accepts the image and returns the location(s) of the detected face(s). Therefore primitive agents are categorized into classes and some class-specific methods are commonly defined. Example of an interface for face detector primitive agent class is as follow:

- *void setImage(base64\_encoded\_data imagecontents):* set input image contents
- *string getFaceLocations():* do face detection, and return face location(s)
- *string setSPAKIP(string ip\_address):* inform the knowledge server at the specified IP address if a face is found

- *base64\_encoded\_data* *getProcessedImage()*: return input image with a rectangle around each face

More technical details on the architecture will be discussed in Chapter 6 in the topic of prototype development.

### 3.6 Summary

This chapter presents an overview of the proposed architecture for robots. The architecture is modular. Robot components are designed as agents. The architecture is hybrid; lower-level tasks are conducted by primitive agents and higher-level tasks are managed symbolically by a special agent knowledge manager. The knowledge manager maintains the knowledge about the world of interest in its knowledge base. It perceives changes in the environment, updates the knowledge base accordingly, and generates output actions. The next two chapters discuss more details of the knowledge manager and development of a robot application based on the architecture. Discussions and comparisons to other work are presented afterwards in Chapter 7.

Future work on agents development includes improvement of primitive agents' wrapper code to be multi-threading and reentrant to support concurrent requests. In such cases that many requests arrive at the same time, task priority should be supported. This includes the ability to pause and resume (or cancel) the current task while processing other higher priority requests.



## Chapter 4

# Frame-based Knowledge Manager

This chapter presents the knowledge manager (KM), a crucial component in the knowledge-based interactive robot. KM maintains the knowledge about the world of interest and its knowledge base and manages robot actions. Frame-based technique is used to represent knowledge in KM because it can represent the world meaningfully and naturally for both robots and human-beings, and is flexible to maintain a variety of knowledge and information. In this work we extended the frame model with new extensions to better support representation of dynamic knowledge and management of robot actions.

This chapter presents the frame model used in our KM, newly proposed dynamic extensions, and the implementation of the concept on the SPAK knowledge platform.

### 4.1 Frame Model

A frame is a data-structure for representing a stereotyped situation [11]. Based on the concept of frame proposed by Minsky in [11], many variations of frame-based systems have been developed. In this work we started from the model used in SPAK [10] and introduced new extensions to it.

In SPAK (and actually in almost all frame-based systems), each frame has a set of *slots* with values (optionally filled with default values). Slot values can be of type scalar, e.g., *string*, *integer*, *real*, array, instance (as a link to other frame instance), or procedure (JavaScript language is supported), and can have a *slot condition* attached. For example, the slot *Length* of the *LongLine* frame shown in Figure 4.1 is of integer type (not shown in the diagram) and has a condition stating that the value must be greater than 0: (*s.Length* > 20)<sup>1</sup>.

Frames inherit properties from their parents, i.e., parent and child have an IS\_A relationship (relationship names will be written in capital letters). The *Line* frame in Figure 4.1 has three children: *LongLine*, *ShortLine*, and *ThickLine*. Instance is a realization of frame(s).

---

<sup>1</sup>In SPAK, the variable *s* means the current frame being evaluated (similar to *this* in Java).

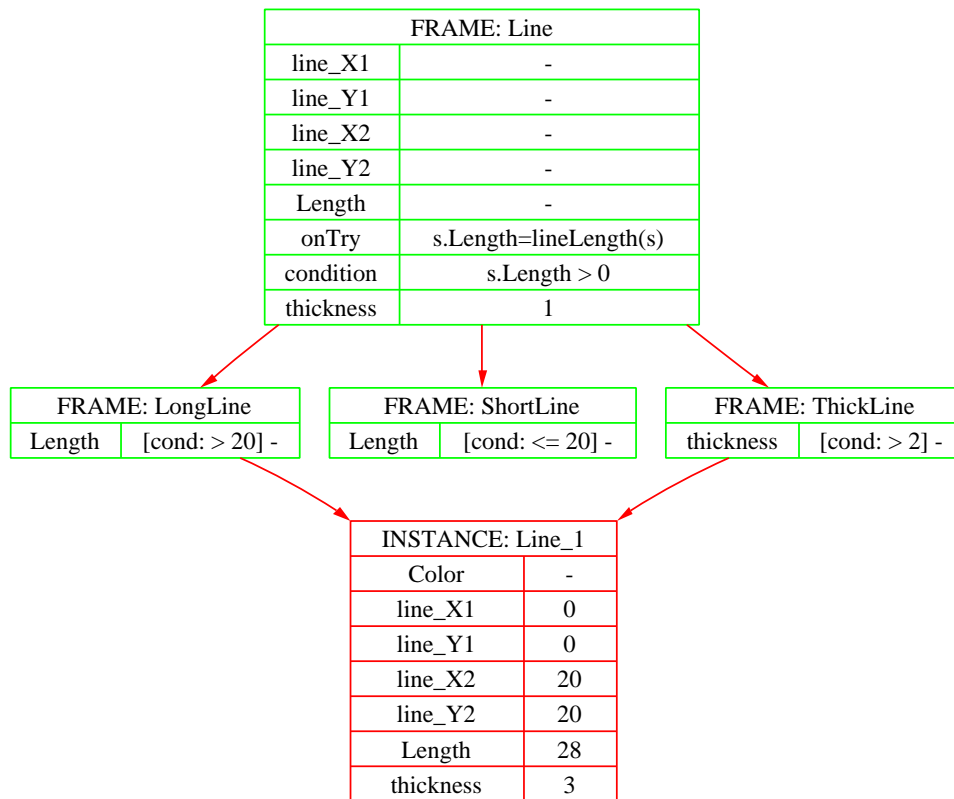


Figure 4.1: Hierarchy of frames representing lines including an instance *Line\_1* as a child of both *LongLine* and *ThickLine* frames. In each frame and instance, its slots are shown in the left column and their values in the right column, optionally with a condition in the square brackets (e.g., the *thickness* slot must be greater than 2 for the *ThickLine* frame). Note that the green tables represent frames and the red ones represent instances. Red arrows are used to indicate IS\_A relationships.

A sample instance *Line\_1* has two parents, *ThickLine* and *LongLine*, as its properties match conditions of both parent frames.

## 4.2 Knowledge Manager Roles

KM is designed to perform three main roles. First, by means of inputs from sensory agents, it maintains representations of the world of interest in the World model. Second, it manages the actions as specified in the Tasks/Actions model. Actions can be updates of knowledge contents or output actions to human and environment. Third, it answers knowledge query from other agents.

Internal mechanisms inside KM are shown in Figure 4.2. KM is usually started with some pre-defined knowledge. Sensory agents detect changes in the world and submit information

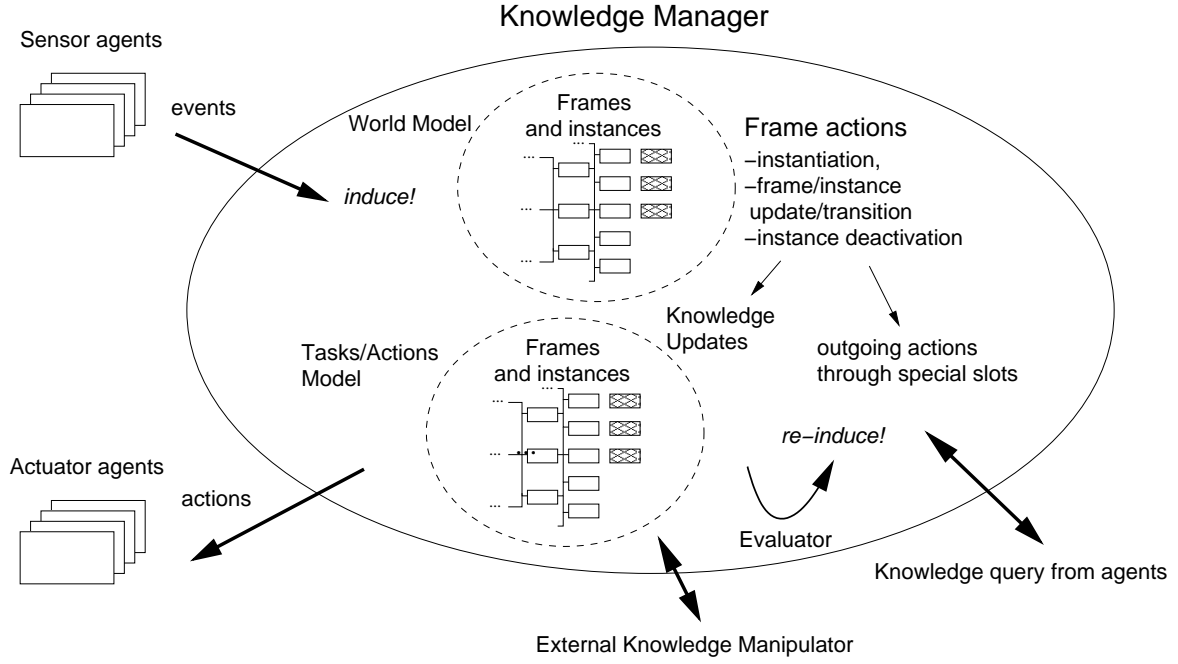


Figure 4.2: Input, output, and actions mechanisms of SPAK knowledge manager

to KM, followed by the *induce* command to trigger the reasoning process. KM reasoning engine is started and it updates the knowledge contents, e.g., the World model, according to the newly received data. The updates can be instantiation, modification, or deletion of frames, frame instances, and their slot values. Along with these updates, some output actions might be generated through the use of special slots in related frames.

Note that although in Figure 4.2 there seems to be a clear distinction between the World and Tasks/Actions models. However, in practice, it is up to the designer to decide how knowledge contents should be arranged.

### 4.3 Frame Model Extensions

By using the conventional frame-based knowledge model in our robotics environment, we encountered some limitations, especially when dealing with changing knowledge. Thus we enhanced it with new dynamic extensions namely, time-based layer, evaluator, frame actions priority, and new special slots and slot flags. In the next two sub-sections, we describe unique features of SPAK, special slots, and slot flags. Then we go on through each extension to the frame model.

Special slot	Details
<i>condition</i>	must be evaluated as true for the instance to exist
<i>priority</i>	indicates the priority of the frame
<i>singleparent</i>	indicates that the frame should have at any time only one parent
<i>needfirstparent</i>	indicates that the first parent (that the frame has since instantiated) must always exist, otherwise this frame will be deactivated
<i>directcreate</i>	indicates that the frame must be directly instantiated only, i.e., other frames can not be reinduced to become children of this frame
<i>onTry</i>	executed when trying to instantiate a frame (forward chaining), must return true, otherwise the instantiation process stops
<i>onBTry</i>	similar to <i>onTry</i> but for backward chaining
<i>onInstantiate</i>	executed when the frame is instantiated
<i>onUpdate</i>	executed when a slot value is updated
<i>onEvaluate</i>	executed when the instance is evaluated
<i>onDestroy</i>	executed when the instance is deactivated
<i>onLoad</i>	executed once when the knowledge manager loads the knowledge contents
<i>onTransition</i>	executed when an instance does not represent its parent frame(s) anymore
<i>onTransitioned</i>	executed when an instance becomes a children of new parent frame(s)

Table 4.1: List of special slots supported in SPAK

### 4.3.1 Special Slots

In our use of KM to manage robot behaviors, output robot actions generated by KM are specified in procedural script (in JavaScript language) in special slots. In other words, special slots can be used to specify possible outgoing actions from the frame. They can be also used to specify frame characteristics like condition to exist and priority.

Some of the SPAK special slots, namely *condition*, *onInstantiate*, *onDestroy*, *onLoad*, have been already discussed in [10]. However, for completeness and ease of understanding, here we describe not only newly added special slots but also those previously introduced. Special slots whose names begin with “on” are on-event or event-driven slots. Their procedural contents will be executed when the corresponding events occur, which then might generate requests to actuator-type agents, causing output actions. Table 4.1 lists all special slots in SPAK knowledge manager.

Details and usages of each special slot are as follows:

#### *condition:*

The special slot *condition* of type “procedural” must be evaluated as *true* for the frame to exist. It can be also used in combination with slot conditions discussed earlier. JavaScript expressions specified in the *condition* slot will be executed and the evaluation result must be true, otherwise the frame can not be instantiated. In case of instance, if its *condition* slot

evaluation becomes false, the instance has to be moved to be a child of the Root frame or other frames whose *condition* can be evaluated as true. If such frames can not be found, the instance will be deactivated.

Frames hierarchy defines the order of frames' conditions evaluation. Conditions are evaluated from the top-most ancestor to the current frame down the knowledge tree in sequential order. However, this can be overridden by manually specifying frames' priority.

***priority:***

The importance level of a frame can be specified by setting an integer value to the *priority* special slot. The higher the value is, the higher the priority the frame has. This will be described more in the subsequent section.

***singleparent:***

If this slot is set to true, the instance will have, at any time, only one parent and will be deactivated if the number of parent becomes zero. This means: 1) When reinducing, if it lost its parent, it will be tried against all parents up the hierarchy. If no suitable parent is found, the instance will be deactivated, 2) When reinducing, if it still has a parent, it will be tried only against more specialized children of that parent.

***needfirstparent:***

If this slot is set to true, the first parent of this instance, i.e., the parent it was first instantiated from, must always exist. If the condition for this parent becomes invalid, the instance is deactivated.

***directcreate:***

This slot indicates that the frame can be instantiated only in the induce process, not the reinduce one. That is, other frames can not be reinduced to become children of this frame.

***onTry:***

The *onTry* slot is executed when the inference engine tries to check whether the frame can be instantiated. For example, the *Line* frame in Figure 4.1 has an *onTry* slot containing procedural script to calculate the line's length from its starting and ending points and update the *Length* slot accordingly. When trying if a frame can be instantiated, the expression in its *onTry* slot is executed. This is useful in case that some calculations are needed before evaluating the condition of the frame. For example, a *LongLine* frame is a *Line* frame with the

length of more than 50 cm. In order to verify this condition when trying to instantiate the frame, the code to calculate of line length is specified in its *onTry* slot.

Also the *onTry* slot can be used to check some conditions when instantiating a frame. If the conditions are not valid, it can put false the return value. The inference engine will stop instantiating the frame. This is similar to the *condition* slot, except that *onTry* is only executed once in the trying phase, while the condition slot is checked at the trying phase as well as during the instance's life by the *reInduce* process and *Evaluator*.

#### ***onBTry:***

In case of backward chaining where it is necessary to know what it takes in order to instantiate a frame, the information on how to find the value of those required slot values can be specified in this *onBTry* (short for on-Backward-Chaining-Try) slot. For example, when booking the bus ticket, it is required that the travel time and destination are known before proceeding with the booking process. To obtain this information, the system can ask a user by making a speech dialogue, for example. These procedures can be specified in the *onBTry* slot.

#### ***onInstantiate:***

Once a frame is instantiated, the procedural code specified in its *onInstantiate* slot is executed. For example, the *onInstantiate* slot of the *Say* Action frame contains the code to send a given text to the text-to-speech agent, which will generate and output the corresponding speech sound to robot's speakers.

#### ***onDestroy:***

Similar to the *onInstantiate*, actions to be done when a frame is deactivated can be specified in the *onDestroy* slot. Note that for the instantiation case, the *onInstantiate* slots of the upper parent frames will be executed first, followed by those of other frames down the hierarchy. But for the deactivation case, it is reverse: the *onDestroy* slot of the current frame itself will be executed first, then traversing up to the parent in the hierarchies.

#### ***onUpdate:***

When a slot value of a frame is changed, the JavaScript code specified in the *onUpdate* slot will be executed. For example in a *Line* frame, when the line starting or ending point changes its position, the line length can be recalculated by putting the appropriate script in the *onUpdate* slot.

***onEvaluate:***

In many cases, some frames need to be updated or some actions need to be done without having any incoming events from outside. The designer can specify in this slot what to do periodically (e.g., recalculate something or check if things are still valid). The Evaluator thread will be started at a certain period, and check and execute this code.

***onLoad:***

When KM starts loading the knowledge contents from a file. The content of this slot (usually we have it in the Root frame) will be executed. This is used mainly for the initialization of variables, and loading of JavaScript functions.

***onTransition:***

When the condition of a certain parent of an instance became invalid (i.e., it is evaluated to false), the instance no longer represents this parent frame and will be removed from the parent frame's children list. Before removing, the contents of the parent frame's *onTransition* slot is executed. With this, knowledge designer can specify what to be done in case of frame parent changing.

***onTransitioned:***

In the reInduce process, when an instance has slots' values that match a certain frame (that it has not been a child of before), generally (see special slots *singleparent* and *needfirstparent* for exceptions) it will become a child of that frame. The contents of that new parent frame's *onTransitioned* slot will be executed.

A sample usage of some special slots is as follows. For the sample *Line* frame shown in Figure 4.1, if we want the line length be recalculated when the line starting or ending point changes, we can add an *onUpdate* slot with the content similar to the *onTry* slot's. When a slot value of a frame is changed, the code in the frame's *onUpdate* slot will be executed. Output actions from the system can be obtained through these special slots of related knowledge frames. When a frame is updated, frames that depend on this frame (i.e., frames that have this instance as a value in their instance-type slots) will be updated as well.

Some special slots are discussed in more details in the following sub-sections.

### 4.3.2 Slot Flags

A frame and its instance slot(s) are usually considered to represent HAS-A relationships, e.g., "a car has four wheels". There are other useful relationships and parameters of relationships.

Slot flag	Stands for	Meaning
R	Required	a valid slot value is required
BO	Beginning Only	a valid slot value is required, but at the beginning only
DF	Do not Fill	do not automatically fill a value for this slot
S	Shared	(instance slot only) the instance value can be shared
U	Unique	the slot value must be unique in all frame instances

Table 4.2: List of slot flags in SPAK

For example, a triangle is composed of three lines, representing a COMPOSED\_OF relationship, which is different from the HAS\_A relationship in that if these lines are missing, the triangle does not exist anymore, while a car without wheels can still be considered a car. In managing robot behaviors, an action can be TRIGGERED\_BY an event. There are cases that once an action is done in response to that event, other actions should not be triggered anymore. Hence, the action is exclusively triggered by that event. This EXCLUSIVENESS can be considered a parameter of relationships.

If we represent these relationships and parameters as frames, there is no limitation on representing any relationships in the knowledge base. However, for convenience, we propose the slot-flag mechanism to specify some often-used relationships. Available flags are shown in Table 4.2.

The *R* flag indicates that this slot must have a valid value in order for this frame to be instantiated, e.g., the *Human* frame has a *birthdate* slot as a required slot. The *BO* flag, only in case that the *R* flag is set, indicates whether the slot is required during the whole instance life (i.e., COMPOSED\_OF) or just at the beginning when instantiating the frame (TRIGGERED\_BY). For example, a *Triangle* frame always requires three *Line* frames during its life, so the *BO* flag should not be set. On the other hand, suppose a *PickObject* action frame has a required instance slot whose value must be a *PickUpRequest* event frame instance. Even if the request frame expires (more detail about this in the next section), the action frame should still exist until the task is accomplished or there is a cancel message from a human. In this case the *BO* flag of the *PickObject* slot should be set.

The *DF* and *S* flags are used in *instance*-type slots only. If the *DF* flag is set, it means that during the induce process, do not find a value automatically for this slot (which the inference engine will normally do – find a valid instance in the system that can match the slot condition). This is useful in case that the designer would like to force specification of the value in order to instantiate the frame. The *S* flag indicates that, for this slot value (which is an instance name), if this instance has already been used in instantiating other frames, it can be used again in this frame or not, i.e., checking the EXCLUSIVENESS. For example, the *Triangle* frame needs instances of *Line* frames in its three slots. If the *S* flag of these slots is not set, these *Line* frames must not have been used in the instantiation process of other frames



(e.g., they can not be part of other triangles or other objects). The *U* flag specifies if the value of this slot must be unique. In case that the *U* flag is set, the new slot value for this slot will result in new instantiation of the frame, otherwise the instance of this frame, if it exists, will be updated instead, i.e., no new instance is created.

### 4.3.3 Time-based Layer

By using the conventional frame-based knowledge model in our robotic environment, we encountered some limitations. First, the conventional frame-based model is found not able to handle temporal information well. Frame hierarchies reflect the structural view of the current world of interest. The current set of instances represent existing things in the world at the current time. Changes in the environment triggers KM to make appropriate changes to its knowledge contents. The knowledge is therefore up-to-date according to the current situation.

However, sometimes it is necessary to access history information. For example, consider a knowledge hierarchy containing a *Human* frame and its sub-class frames: *Professor*, *Associate Professor*, *Lecturer*. The robot met a human named *John*, learned that he was a lecturer, and added the information into its knowledge. Some years later *John*'s position changed from lecturer to associate professor and finally to professor. With proper updates every times *John*'s position changed, the robot could answer questions like "*What is the position of John now?*", but it would have a hard time answering questions like "*What was the previous position of John?*" and "*When did John get promoted from lecturer to associate professor?*". Keeping track of temporal information is crucial in order to answer such questions.

In robot applications, temporal information is crucial. Frames representing events or states of the world are continuously updated to reflect the changes. Some conditions for frame actions depend not only on the current frames and slots but also on the past ones.

We propose that this temporal information should be properly handled by keeping track of all changes in the knowledge contents. Frame instances, which can change over time, become layers of snapshots of a frame at each point in time, stacking on top of each other.

One can argue that we can simply create a special slot *PreviousPosition* in the *Human* frame to keep the old status information and probably another slot for keeping the time when the status changes, or even treat *John*'s human frames with different positions as different frames. However, with this, we will end up with either having an overwhelming number of slots for every information changes we would like to keep track of, or having extremely large amount of frame instances in the system.

Time-based layer support in SPAK is illustrated using the previous example of Mr. John in Figure 4.3. SPAK KM provides methods to access this history data, e.g., the old slot values with the modification time, old parent frames and the instance age.

When the value of a slot is changed, the old value is added to a slot value history vector.

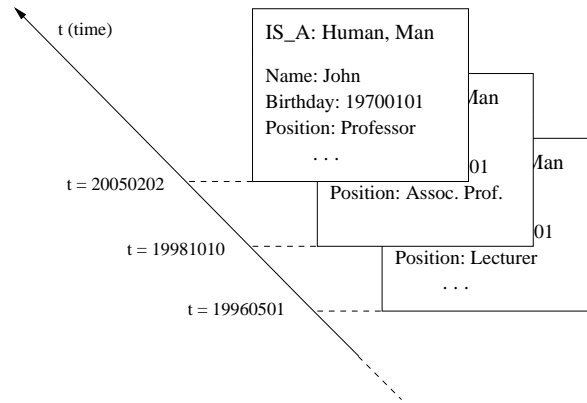


Figure 4.3: Snapshots of the frame instance representing *John* at various time points

Old slot values can be retrieved by calling the function  $KFrame::getSlotValue(sl, t)^2$ , which will return the value of slot  $sl$  at an absolute time  $t$  or past time  $t$  if  $t$  is negative. Frame instantiation time is also kept and the age of a frame can be retrieved by calling the function  $KFrame::getAge()$ .

With this time-based layer, we can create robot behaviors that vary depending on the history. For example, the current value of the slot *length* of a frame  $S$  is  $S.getSlotValue("length")^3$ . That value at 10 seconds ago can be obtained by calling  $S.getSlotValue("length", -10)$ . It is also possible to specify the frame's condition based on time. For example, we want a robot behavior: *Do Action A if the face has been present since 2 seconds ago*. Assume that *myFace* is an instance of a *Face* frame, and *status* is a slot of *Face* frames indicating the status of the face (absent or present) from the face detector agent. The condition slot of the action frame  $A$  can be set as follow:

$myFace.getSlotValue("status") == "present" \ \&\& \ myFace.getSlotValue("status", -2) == "present"$

#### 4.3.4 Evaluator

Over time, some frames need to be updated and some actions need to be done without any triggering events from outside. The simplest example is human age, which increases over time. Frames that represent tasks or actions can also be updated so that they generate different actions at different times. This gives the idea of active frames, where frames themselves are not only used to store knowledge and waiting for updates by external agents, but they can be active by themselves.

We propose that the knowledge base must provide mechanisms to accommodate this idea. KM has a special process called *Evaluator*, which is executed periodically by the system.

<sup>2</sup>A frame or an instance in SPAK is represented by a KFrame Java class.

<sup>3</sup>As a notation used in SPAK, the symbol  $S$  and  $s$  means the current frame represented by KFrame or KFrameScript objects accordingly.

The knowledge designer can add procedural code in the *onEvaluate* special slot to specify what is to be done when it is evaluated. The Evaluator process will check and execute the contents of this slot.

When a frame is being evaluated, its slots' conditions and its *condition* special slot are also checked with those of its parent frames whether they are still valid or not. Should they become invalid, which means the instance no longer represents the parent frame, the inference engine will remove the IS\_A connection to that parent frame, and find out what other frames it can be a child of. If it can not be a child of any frames, it will be moved to be a child of the *Root* frame (top of the hierarchy). In the case that the condition is still valid, the frame will be checked further if it can be more specific (e.g., *Child* or *Adult* instead of *Human*). In the case of a frame parent changing, the content of another special slot *onTransition* will be executed.

#### 4.3.5 Priority Support for Frame Actions

Incoming events and clock ticks can trigger actions in KM, e.g., instantiation, slot modification, and time-based evaluation. In some cases, there is more than one candidate frame to be processed at the same time, all caused by the same event. The question is, which frame should be processed first? The order is important, since a frame action can change the knowledge contents itself. It might even invalidate the condition of another frame that is about to be processed at the same time.

We propose that it should be possible to assign a priority value in each frame in order to indicate which frame would win in the case that many frames are simultaneously qualified for frame actions. This can be manually assigned in each frame, or defined as a general policy.

The design of SPAK adopts a general policy that gives higher priority to more specific frames, i.e., frames that are further away from the *Root* frame. If some frames are equally specific, the oldest frame wins, to prevent newer frames, which might not have been thoroughly tested, from overriding old behaviors unintentionally. If a newer frame should precede the older, then it should be explicitly defined by specifying the value of the special slot *priority* manually or the older one should be deleted first. This is also to maintain the system consistency and ease of design, e.g., in case that certain data can trigger instantiation of many frames, the system should always choose the oldest one unless specified otherwise, if not, the system would keep changing its behavior, resulting in difficult design process. However this issue depends much on the designer. This general policy can be overridden anyway by using the special slot *priority*. With this mechanism, the knowledge designer can control the order of frame actions as desired.

As an example, imagine a robot that is programmed to normally say "Good bye" to the human when he is leaving the laboratory at the end of the day. However, in the case it knows

that it is going to rain this evening and notices that he forgot to take his umbrella with him, another action to warn him like “*It is going to rain this evening, would you like to take your umbrella?*” should have higher priority and be done first.

## 4.4 SPAK Knowledge Platform

SPAK is a modern software platform for knowledge processing and coordination of tasks. First introduced in [10], SPAK features multiple-inheritance frame-based knowledge management with forward and backward chaining inference engines. It is written in Java, and has a GUI knowledge editor for manual manipulation of the knowledge contents, and a network interface allowing collaborations with other agents.

SPAK was originally inspired by the ZERO++ frame-based knowledge engineering environment used in the HARIS project [56]. The desired features of SPAK are *platform-independent* as existing robots and software modules often rely on different platforms or OS'es, *network-aware* as the modules must interact on a network with other agents, and *user-friendly* so that developers can intuitively access the system.

The current version of SPAK consists of the following software components:

- **GUI User Interface:** A user-friendly Graphical User Interface (GUI) to the internal knowledge base and the inference engines. It provides users direct access to the frame-based knowledge hierarchies.
- **Knowledge Base:** The core module of SPAK, which maintains the frame systems as Java class hierarchy and performs knowledge conversion to/from XML format for exporting knowledge data to other applications.
- **Inference Engines:** The engine to verify and process information from external modules, which may result in instantiation, modification, and deactivation of frame instances in the knowledge base, and execution of actions specified in related special slots.
- **JavaScript Interpreter:** An interpreter for JavaScript [106] code used for defining condition and procedural slots in a frame. It provides an access to a rich set of standard Java class libraries that can be used for developing SPAK applications.
- **Network Gateway:** A daemon program allowing networked software agents to submit new slot-value pairs information to the knowledge base and to access the knowledge stored in SPAK.

#### 4.4.1 Graphics User Interface (GUI)

The graphical user interface of the SPAK knowledge editor is shown in Figure 4.4. All basic knowledge manipulation tasks can be performed via the interface. The left window displays the current knowledge frame hierarchies. Each frame is represented as a click-able button. These frame buttons are linked with red lines indicating IS-A relationships among them. Clicking on the frame button brings up a new frame property window showing the frame's slots. For example, a property window of the *Line\_1* frame instance is shown in Figure 4.5.

#### 4.4.2 Knowledge Base

This section describes the definition of frames hierarchy inside SPAK, and the specification of frame and slot.

##### Frames and Instances

Frames and their instances are represented as objects of Java class *KFrame* inside SPAK. Each object contains two sets of pointers pointing to other frame objects in order to form IS-A relationships. One set points to all the children, the other points to the parents. The frame can have multiple parents in case of multiple-inheritance. All *KFrame* objects contain a property flag indicating if the object is a frame (a subclass of its parents), or an instance (of its parents). In case of an instance, the slots are copied from all its parents and values are filled in. In case of a frame, there can be only a small number of slots which augment or override the slots of its parents.

Type	Description
<i>String</i>	An arbitrary character string.
<i>Integer</i>	An integer numeric value.
<i>Real</i>	A real number value.
<i>List</i>	An array of slot objects of any types.
<i>Instance</i>	An instance of a frame. The <i>Argument</i> field must contain the name of that frame.
<i>Procedure</i>	A storage for arbitrary JavaScript expressions, which can be called as a method in JavaScript context.

Table 4.3: Supported slot types in SPAK

##### Slots

Included in the frame object is a set of slots. Frame slots can be considered member variables or member functions of a *KFrame* object, depending on the slot type. Slot type determines how the slot value can be accessed in JavaScript context. For example, a procedural slot can

be called as a JavaScript method; while data type slots can be accessed as member variables. Some properties of a slot are: *Name*, *Type*, *Value*, *Condition*, and *Argument*.

A slot can be accessed via its *Name*. Currently SPAK supports slot of types: *String*, *Integer*, *Real*, *List*, *Instance*, and *Procedure*, as listed in table 4.3. For a slot of type *String*, *Integer*, and *Real*, a comparative condition, together with its argument(s), can be applied to the slot value. The slot value must satisfy the condition specified here for an instance of this frame to be created.

SPAK supports all special slots and slot flags as discussed in Section 4.3. Extensions to the frame model proposed in Section 4.3 were added to SPAK. SPAK supports special slots as listed in the previous section. Time-based layer is supported and programming interfaces to access history information are provided. Frame-actions priority is supported in SPAK via the use of the special slot *priority*. The default policy gives priority to instances with longer distance from the *Root* frame.

### Example Knowledge Contents

A sample screenshot of SPAK is shown in Figure 4.4. The right part displays console message of input and output information to and from SPAK. The left part displays the knowledge contents, in this example, a hierarchy of frames representing geometry shapes like line, parallel lines, and other objects. The *Line* frame has children of *LongLine*, *ShortLine*, *ThickLine*, and *Line\_1* instance. This knowledge frames is a SPAK realization of the knowledge hierarchy shown previously in Figure 4.1.

As we can see from Figure 4.4, the *Line\_1* instance is a direct child of both *LongLine* and *ThickLine* frames. When clicking at the rectangle represent *Line\_1* instance, a SPAK window showing its property (slots list and values) is brought up, as shown in Figure 4.5. Each row represents a slot, with columns showing slot type, value, condition, argument (in case the condition is not null), and slot flags R, BO, DF, S, U. Slots which begin with “\_” are used internally by SPAK, e.g., the slot *\_ID* is used to store a unique frame/instance ID, the *\_ISA* slot is used to store current parents of this frame/instance, and the *\_WASA* slot is used to store old parents of this frame/instance.

Another example is shown in Figure 4.6. A hierarchy of frames represents fictitious humans with classification into subframes representing adult, child, etc. The corresponding knowledge tree is shown in Figure 4.7. We can see that the frame *Human* has children of *Child*, *Adult*, and *30up* with different conditions that check the value of the *age* slot. Note that the checking can be done by either using the slot condition (e.g., *age*<15 and *age*>30) or using the special slot condition (e.g., *s.age* >=20).

The property of the *Child\_1* instance is shown in Figure 4.8. Note that the condition of its parent *Child* frame requires that the slot *age* must be less than 15 (otherwise it cannot be considered a *Child* frame instance). In the *onEvaluate* slot of the *Human* frame, there is a

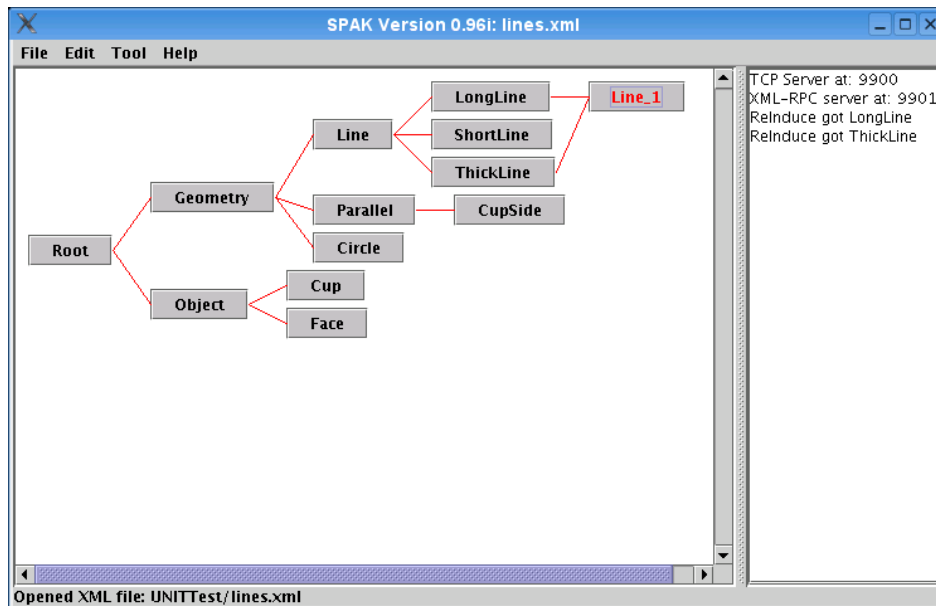


Figure 4.4: A screenshot of SPAK loaded with the knowledge contents representing line frames from the example in Figure 4.1

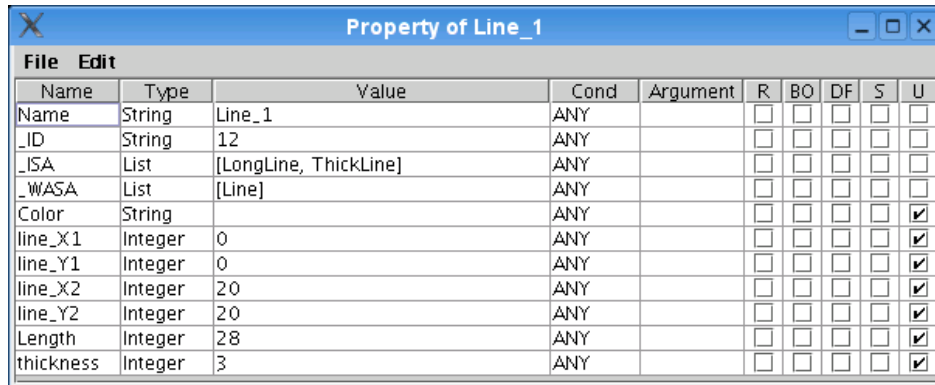
JavaScript expression to increase the value of the *age* slot by one to simulate the increasing of human age. The Evaluator thread will check this *onEvaluate* slot and execute it. Thus this *Child\_1* frame instance will have an increasing age over times and will be moving from an instance of the *Child* frame to the *Human*, *Adult* and finally the *30up* frame.

### Knowledge Storing

SPAK stores the knowledge contents in XML. XML is an open and standardized storage format. Hence it is easy to share the knowledge with other applications. In SPAK, the knowledge frame hierarchies are serialized into the text-based XML format for storing in local file storage or exporting to other applications.

Example of an XML-encoded knowledge content from the human and children frames example shown in Figure 4.6 is as follow:

```
<?xml version='1.0' encoding='utf-8'?>
<FRAMELIST>
  <FRAME>
    <NAME>Root</NAME>
    <ISINSTANCE>FALSE</ISINSTANCE>
    <SHOWCHILDREN>TRUE</SHOWCHILDREN>
    <SLOTLIST>
      <SLOT>
        <NAME>onLoad</NAME>
        <TYPE>TYPE_STR</TYPE>
```



Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	Line_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	12	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[LongLine, ThickLine]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[Line]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Color	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
line_X1	Integer	0	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
line_Y1	Integer	0	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
line_X2	Integer	20	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
line_Y2	Integer	20	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Length	Integer	28	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
thickness	Integer	3	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 4.5: A screenshot of a property window of the *Line\_1* instance in Figure 4.4. Each row shows a slot with its name, type, value (frame default value), slot condition, argument (of the condition), and slot flags (R, BO, S, U).

```

<CONDITION>COND_ANY</CONDITION>
<ARGUMENT></ARGUMENT>
<VALUE>load( "ottbot2/common.js" )</VALUE>
<REQUIRED>FALSE</REQUIRED>
<REQUIREDB>FALSE</REQUIREDB>
<DONTFILL>FALSE</DONTFILL>
<SHARED>FALSE</SHARED>
<UNIQUE>FALSE</UNIQUE>
</SLOT>
</SLOTLIST>
</FRAME>
<FRAME>
  <NAME>Human</NAME>
  <ISA>Root</ISA>
  <ISINSTANCE>FALSE</ISINSTANCE>
  <SHOWCHILDREN>TRUE</SHOWCHILDREN>
  <SLOTLIST>
    <SLOT>
      <NAME>age</NAME>
      <TYPE>TYPE_INT</TYPE>
      <CONDITION>COND_ANY</CONDITION>
      <ARGUMENT></ARGUMENT>
      <VALUE></VALUE>
      <REQUIRED>TRUE</REQUIRED>
      <REQUIREDB>FALSE</REQUIREDB>
      <DONTFILL>FALSE</DONTFILL>
      <SHARED>FALSE</SHARED>
      <UNIQUE>TRUE</UNIQUE>
    </SLOT>
    <SLOT>
      <NAME>onEvaluate</NAME>
      <TYPE>TYPE_STR</TYPE>
      <CONDITION>COND_ANY</CONDITION>
      <ARGUMENT></ARGUMENT>
      <VALUE>s.age++</VALUE>
    </SLOT>
  </SLOTLIST>
</FRAME>

```



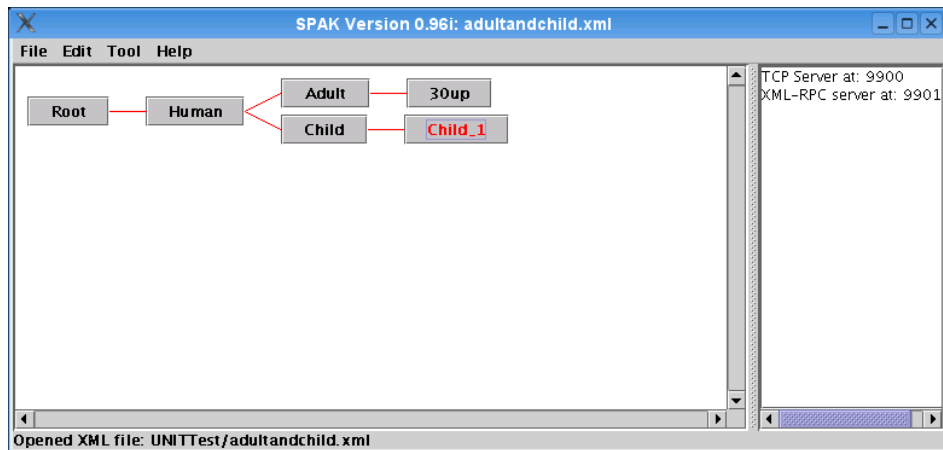


Figure 4.6: A screenshot of SPAK showing the knowledge of the *Human* frame and its children

```

    <REQUIRED>FALSE</REQUIRED>
    <REQUIREDB>FALSE</REQUIREDB>
    <DONTFILL>FALSE</DONTFILL>
    <SHARED>FALSE</SHARED>
    <UNIQUE>FALSE</UNIQUE>
  </SLOT>
</SLOTLIST>
</FRAME>
<FRAME>
  <NAME>Adult</NAME>
  <ISA>Human</ISA>
  <ISINSTANCE>FALSE</ISINSTANCE>
  <SHOWCHILDREN>TRUE</SHOWCHILDREN>
  <SLOTLIST>
    <SLOT>
      <NAME>condition</NAME>
      <TYPE>TYPE_STR</TYPE>
      <CONDITION>COND_ANY</CONDITION>
      <ARGUMENT></ARGUMENT>
      <VALUE>s.age >= 20</VALUE>
      <REQUIRED>FALSE</REQUIRED>
      <REQUIREDB>FALSE</REQUIREDB>
      <DONTFILL>FALSE</DONTFILL>
      <SHARED>FALSE</SHARED>
      <UNIQUE>FALSE</UNIQUE>
    </SLOT>
  </SLOTLIST>
</FRAME>
<FRAME>
  <NAME>30up</NAME>
  <ISA>Adult</ISA>
  <ISINSTANCE>FALSE</ISINSTANCE>
  <SHOWCHILDREN>TRUE</SHOWCHILDREN>
  <SLOTLIST>

```

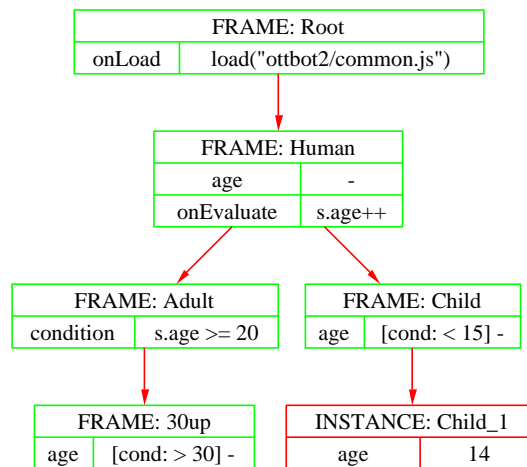


Figure 4.7: Hierarchy of frames representing humans. The red lines represent HAS\_A relationships.

Property of Child_1									
File Edit									
Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	Child_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	9	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[Child]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
age	Integer	14	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 4.8: Property of the *Child\_1* instance

```

<SLOT>
  <NAME>age</NAME>
  <TYPE>TYPE_INT</TYPE>
  <CONDITION>COND_GT</CONDITION>
  <ARGUMENT>30</ARGUMENT>
  <VALUE></VALUE>
  <REQUIRED>TRUE</REQUIRED>
  <REQUIREDB>FALSE</REQUIREDB>
  <DONTFILL>FALSE</DONTFILL>
  <SHARED>FALSE</SHARED>
  <UNIQUE>FALSE</UNIQUE>
</SLOT>
</SLOTLIST>
</FRAME>
<FRAME>
  <NAME>Child</NAME>
  <ISA>Human</ISA>
  <ISINSTANCE>FALSE</ISINSTANCE>
  <SHOWCHILDREN>TRUE</SHOWCHILDREN>
  <SLOTLIST>
    <SLOT>

```

```

    <NAME>age</NAME>
    <TYPE>TYPE_INT</TYPE>
    <CONDITION>COND_LT</CONDITION>
    <ARGUMENT>15</ARGUMENT>
    <VALUE></VALUE>
    <REQUIRED>TRUE</REQUIRED>
    <REQUIREDB>FALSE</REQUIREDB>
    <DONTFILL>FALSE</DONTFILL>
    <SHARED>FALSE</SHARED>
    <UNIQUE>FALSE</UNIQUE>
  </SLOT>
</SLOTLIST>
</FRAME>
<FRAME>
  <NAME>Child_1</NAME>
  <ISA>Child</ISA>
  <ISINSTANCE>TRUE</ISINSTANCE>
  <SHOWCHILDREN>TRUE</SHOWCHILDREN>
  <SLOTLIST>
    <SLOT>
      <NAME>age</NAME>
      <TYPE>TYPE_INT</TYPE>
      <CONDITION>COND_ANY</CONDITION>
      <ARGUMENT></ARGUMENT>
      <VALUE>14</VALUE>
      <REQUIRED>FALSE</REQUIRED>
      <REQUIREDB>FALSE</REQUIREDB>
      <DONTFILL>FALSE</DONTFILL>
      <SHARED>FALSE</SHARED>
      <UNIQUE>TRUE</UNIQUE>
    </SLOT>
  </SLOTLIST>
</FRAME>
</FRAMELIST>

```

### 4.4.3 Inference Engines

SPAK has two inference engines for forward and backward chainings:

- **Forward chaining:** Forward chaining is usually used when a new fact is added to the knowledge base and we want to generate its consequences, which might result in new other facts.
- **Backward chaining:** For a certain thing we want to prove, the backward chaining inference finds implication facts that would allow us to conclude it. It is used for finding all answers to a question posed to the knowledge base.

For example, the forward chaining is used in the event-driven scenario of a robotic application. When the user pushes a switch on the robot, an instance of a *Switch* frame is created, and the scripts embedded in the *onInstantiate* slot are executed.

In contrary, backward chaining is used in such a case as image processing. For instance, if the camera agent is asked to locate a cup in the image, the frame definition of a cup may contain an oval shape, and a pair of parallel lines as required components. This forces the backward chaining engine to try to search for an oval shape, and parallel lines before deciding if a cup exists in the image. Similarly, it can also be used in robot task planning as well, which will be described in Section 4.5.2.

#### 4.4.4 JavaScript Interpreter

JavaScript is used as the scripting language for specifying frame conditions and actions. It is also used to implement additional function libraries required by applications. The library can be loaded together with the knowledge hierarchy by adding a “load” command in the *onLoad* special slot of the knowledge root node. SPAK uses an external library *Rhino* [107] to interpret the scripts. Inside the JavaScript context, SPAK provides a number of classes and methods for manipulating the knowledge frames.

#### 4.4.5 Network Gateway

SPAK is network-accessible through its network gateway module. The module accepts command for submitting slot values to SPAK, starting inference process, and querying and manipulating the knowledge frame hierarchies. Agents can enter a Javascript code to be executed by SPAK inference engine and obtain the result.

SPAK accepts network access via its direct TCP-input (port 9900) and XML-RPC (port 9901) interfaces. A sample access to the TCP port is as follow:

```
$ telnet localhost 9900
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
help
--HELP--
help: This message
frames: List all frame names and slots
Frame-Slot=value: Enter a slot value
Instancename-Slot=value: Modify a slot value of an instance
list: Show current slot buffer
induce: Try to instantiate frame from given slot value(s)
reinduce: Try to reinduce the existing instance(s)
remove: Try to remove instance with given slot value(s)
show: Show instances with given slot value(s)
removeall [framename]: Remove all instances of type Framename
reset: Reset the induction engine
$ xxx: Run JavaScript code xxx, e.g., $ i=1;i
# xxx: Comment Line
--ENDHELP--
```

A remote agent can connect to this port and input information for the inference engine in the format of *Frame-Slot=Value*. For example, the input of *SpeechRecognized-text=Yes* means that the value of the slot *text* of the frame *SpeechRecognized* is *Yes*, which might cause an instantiation of the *SpeechRecognized* frame (if the condition is satisfied).

Slot values of existing instances can be also modified by inputting in the format of *Instance-Slot=Value*. For example, by entering *SpeechRecognized\_1-text=Yes*, it means that the value of the slot *text* of an instance *SpeechRecognized\_1* should be set to *Yes*. In this case an instance name *SpeechRecognized\_1* must exist, otherwise SPAK will treat this input as a new *Frame-Slot=Value* information.

Inputting JavaScript to query and manipulate the knowledge contents via the network is also possible by prefixing the input text with \$. For example, to query the age of the *Child\_1* instance, one can do as follow:

```
$ telnet localhost 9900
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
$ var child = Root.findFrame("Child_1");
$ child.getSlotValue("age");
14
```

## 4.5 SPAK Reasoning Mechanism

The reasoning mechanism of SPAK is based on a simple and natural concept that **frame and instances integrity must be validated at all time**. In other words, the forward chaining mechanism ensures that, when a new information arrives at the system (e.g., via the network or the GUI interfaces), all knowledge frames will be checked if any frames or instances should be created, modified, or deactivated. This mechanism allows SPAK to incorporate new facts to the knowledge base according to information from other networked software agents. The consequence can be also that actions that are defined in special slots like *onInstantiate*, *onUpdate*, *onTransition*, *onTransitioned*, *onDestroy* slots will be executed.

SPAK's forward chaining inference engine can be started by supplying the command "induce" via its network gateway, JavaScript code, or the *Tool* menu of the SPAK GUI editor. This *induce* process will use the available slot-value information (received from other agents via the network gateway) to find out if it can induce or update any new instances. After that, to make sure that the created or updated instances are in a valid condition, another process called *reInduce* checks existing instances if their conditions are still valid, i.e., whether it should be given a new parent or its existing parent should be removed. Algorithms for Induce and reInduce processes are shown in a flowchart in Figure 4.9. Normal arrow lines show process flow and dashed lines represent data flow.

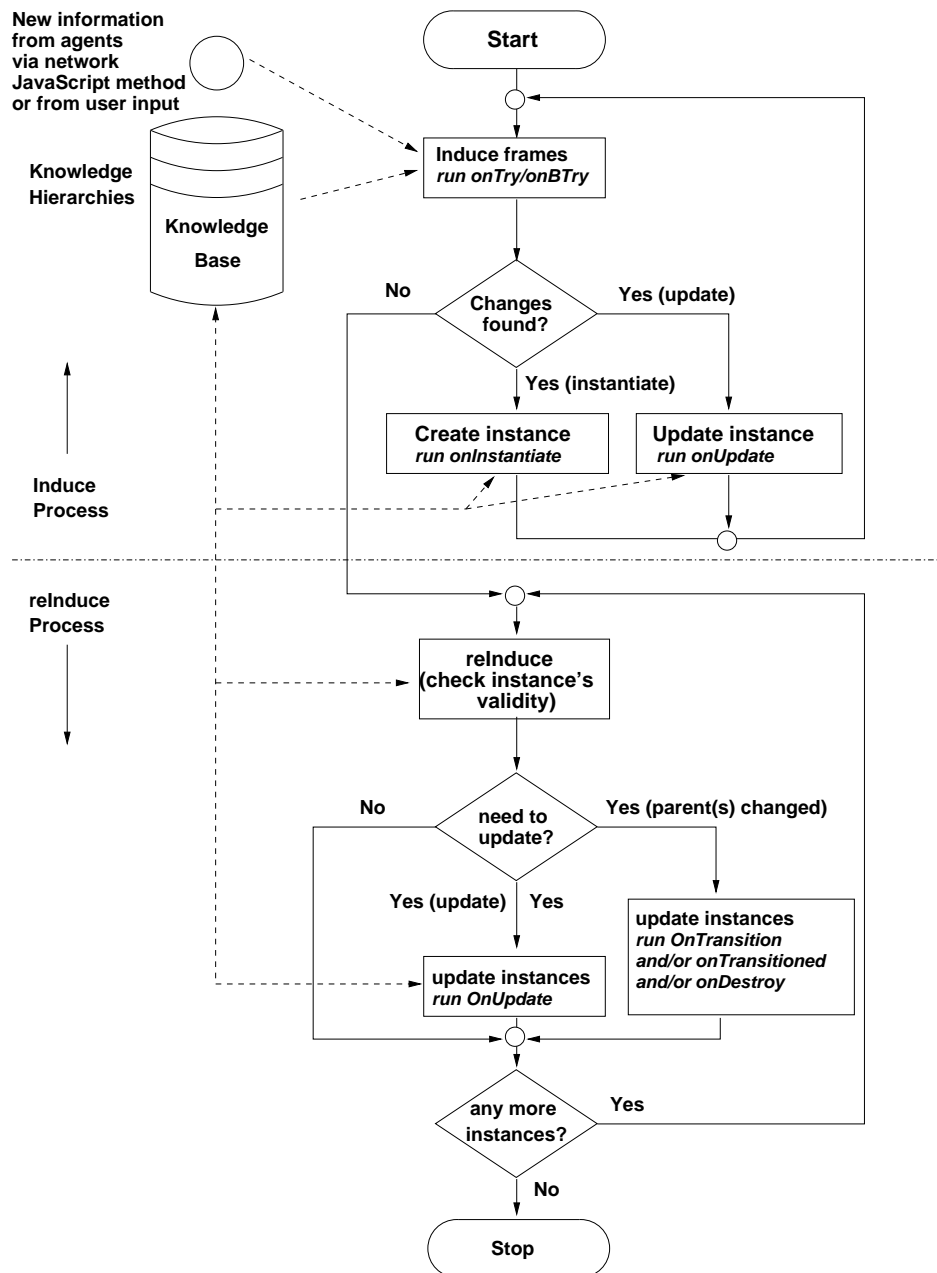


Figure 4.9: A flowchart showing algorithm of the SPAK induce and reInduce inference processes

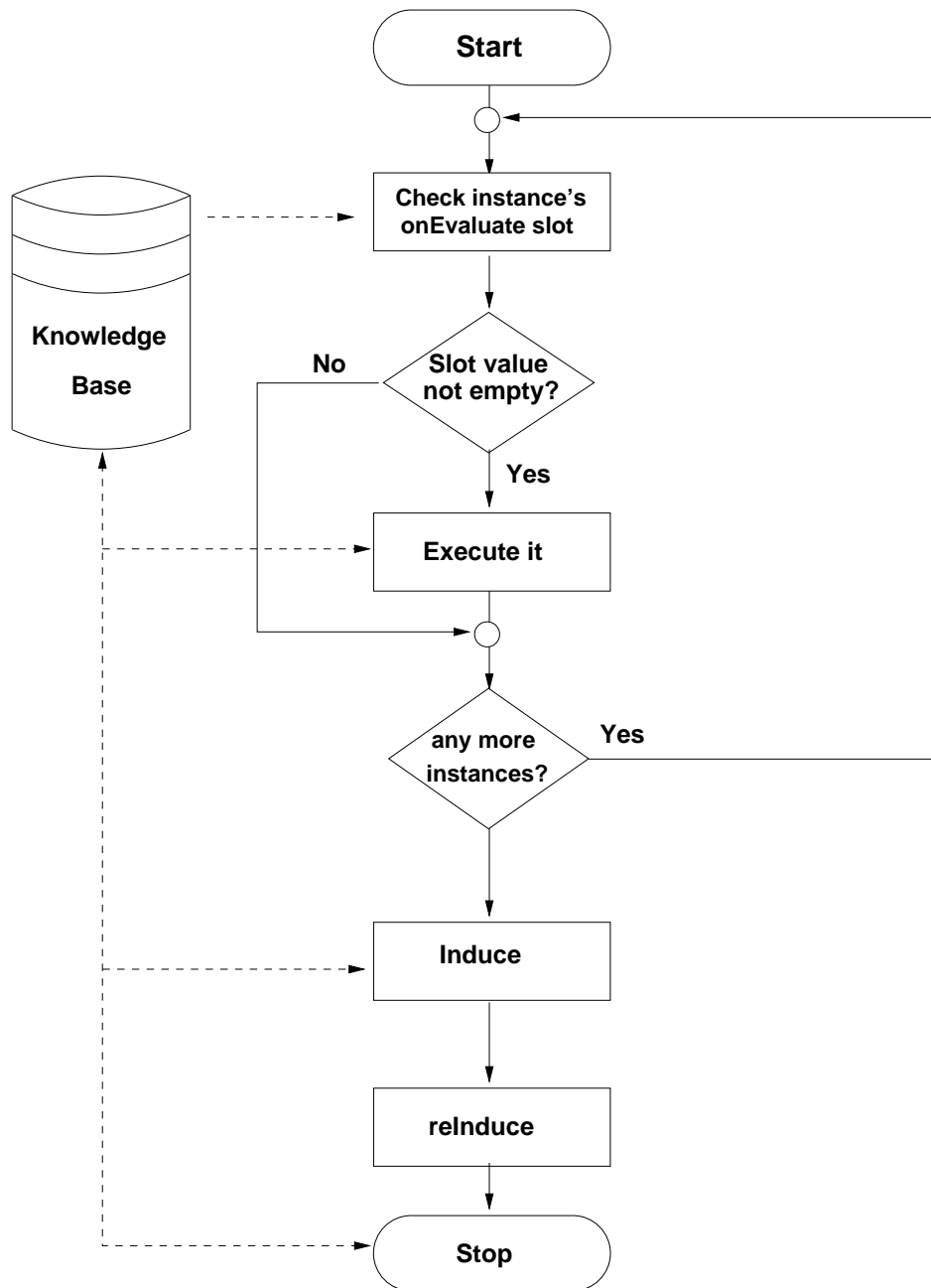


Figure 4.10: A flowchart showing algorithm of SPAK induce and reInduce inference process

SPAK has an *Evaluator* thread that can be either set to run periodically (e.g., every 3 seconds) or in one-shot (for debugging). As shown in the flowchart in Figure 4.10, when the Evaluator thread is started, it checks all instances and executes their *onEvaluate* slots, if exist. Then, as there might be changes in the knowledge contents, it makes sure the integrity of the knowledge base by launching an *Induce* process followed by a *reInduce* process.

Through the backward-chaining inference, SPAK can tell what are the required slots for

a frame to be instantiated, and prompt the user or other network agents to provide information regarding those slots. This can be used in applications like an expert system providing diagnosis of a disease given the user symptoms.

The following subsections briefly discuss how this reasoning mechanism can be used to implement some robot applications.

#### 4.5.1 Scene Understanding

It is necessary that human-interacting robot be able to understand surrounding environment. One way to achieve this is by submitting the images captured by a video camera to a number of image processing modules such as geometry shape detection, facial detection, and facial recognition, as shown in Figure 4.11. Then the results from these modules can be processed by the frame-based knowledge module to identify the objects in the scene.

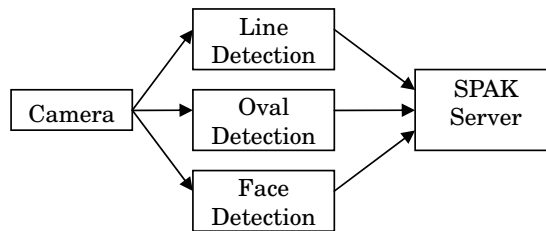


Figure 4.11: Scene understanding with image processing modules and SPAK

Frames of physical objects and their features can be created on SPAK server to perform this task. We reuse again the example frames hierarchy shown in Figure 4.4. Its corresponding knowledge tree is shown in Figure 4.12. Various frames are used to represent geometry shapes like line (*Line*), parallel lines (*Parallel*), cup's side (*CupSide*), circle (*Circle*), and objects like cup (*Cup*) and face (*Face*).

The *Line* frame contains slots like *line\_X1*, *line\_Y1*, *line\_X2*, *line\_Y2* (representing coordinates of its starting and ending points), *Length*, and *Thickness*. A *Parallel* is defined as two lines whose properties meet a certain *condition*. A *CupSide* is a *Parallel* with small difference in line lengths. The frame representing a *Cup* object requires instances of a *Cupside*, and a top *Circle*.

The geometry detection modules keep looking for objects they are responsible for, e.g., "line", "oval", and "face". If they find any, they send information about the object to SPAK. Based on this information, the co-ordinates of these objects and their relationships will be checked and the corresponding frames will be created in SPAK. This new knowledge about objects in the scene can then be used further by other parts of the system.



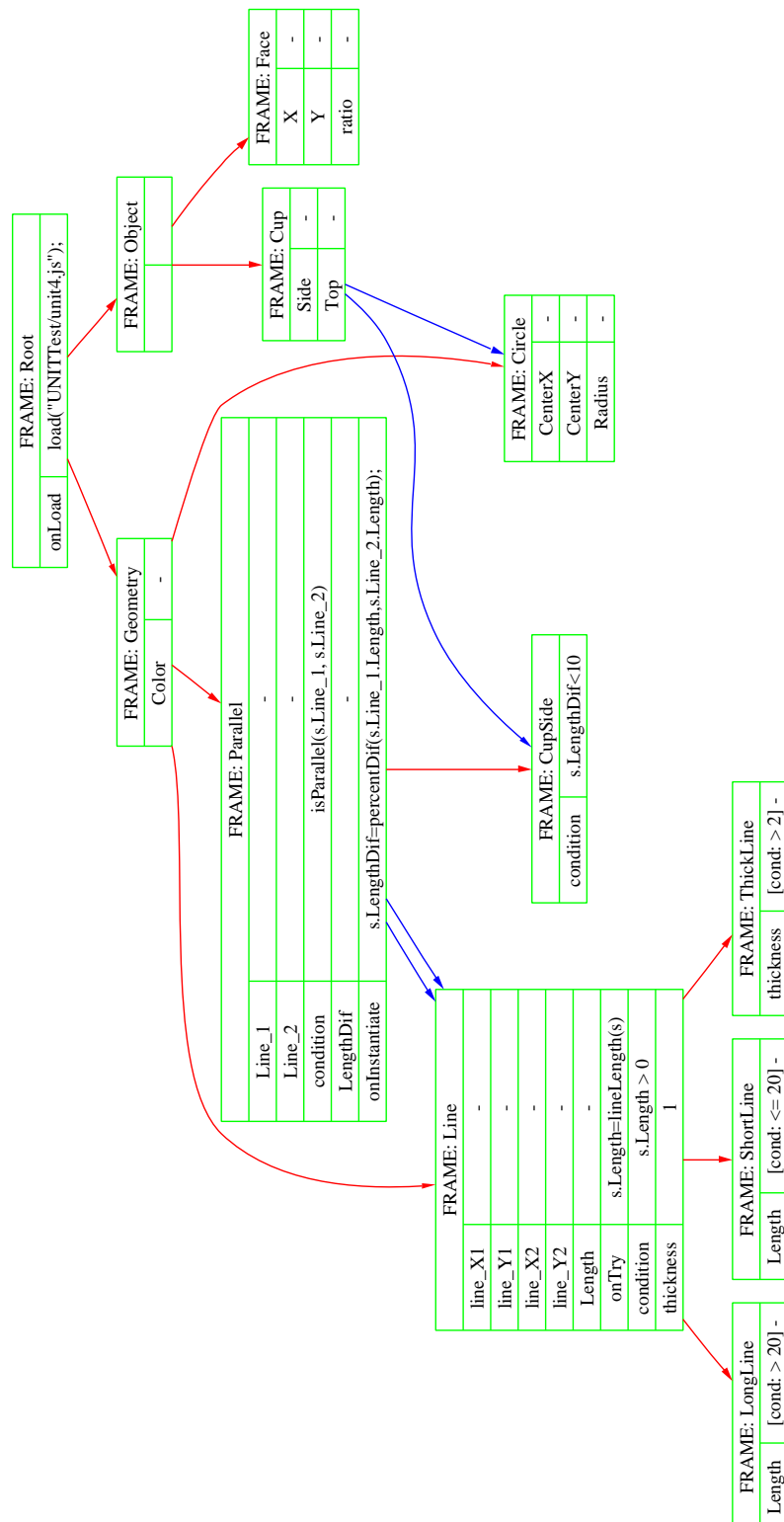


Figure 4.12: A frames hierarchy corresponding to the knowledge contents shown in Figure 4.4. The red and blue lines represent IS\_A and HAS\_A relationships respectively

### 4.5.2 Robotic Tasks Planning

A robot action can often be subdivided into a number of small tasks. For example, if we want the robot to *move* a cup, the robot hand must *reach* to it and *grasp* it prior to *lifting* it up. After that the *moving*, *putting*, and *releasing* tasks can be followed [55].

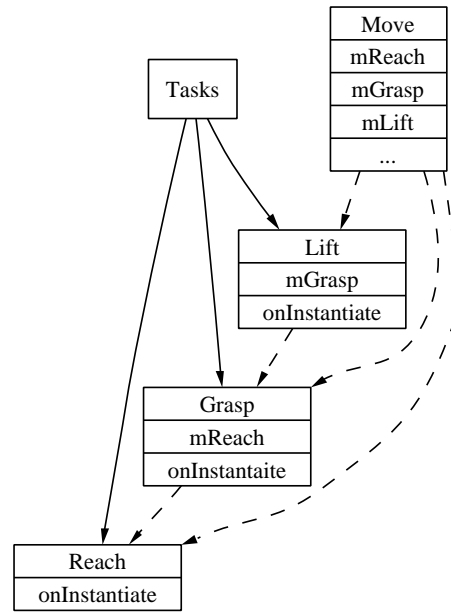


Figure 4.13: Task scheduler for move-a-cup action

The subdividing of an action into tasks and relating these tasks with dependency can be implemented in SPAK frames system by setting the *R* (*required*) flag of instance slots among them. This *R* slot flag which specifies information needed for instantiating a frame instance can be used to establish a dependency among the frames. For example, the *move* action frame require instances of all task frames. A *Lift* task requires an instance of a *grasping* task, which requires further an instance of a *reach* task. This dependency is shown in Figure 4.13.

When the user initiates a *move* action, the backward chaining will check all requirement dependency and make sure that the *reach*, *grasp*, and *lift* tasks are instantiated and executed in the correct order before the action can be declared completed. Commands for instructing the robot to perform each task can be embedded in *onInstantiate* slot of that task frame.

## 4.6 SPAK Knowledge Design Policy

A frame in SPAK can have a life cycle as follows. As shown in Figure 4.2, the *induce* command following incoming events from agents will cause SPAK to start the *induce* process, which will check if there should be any **instantiation** or **updates** of knowledge frames.

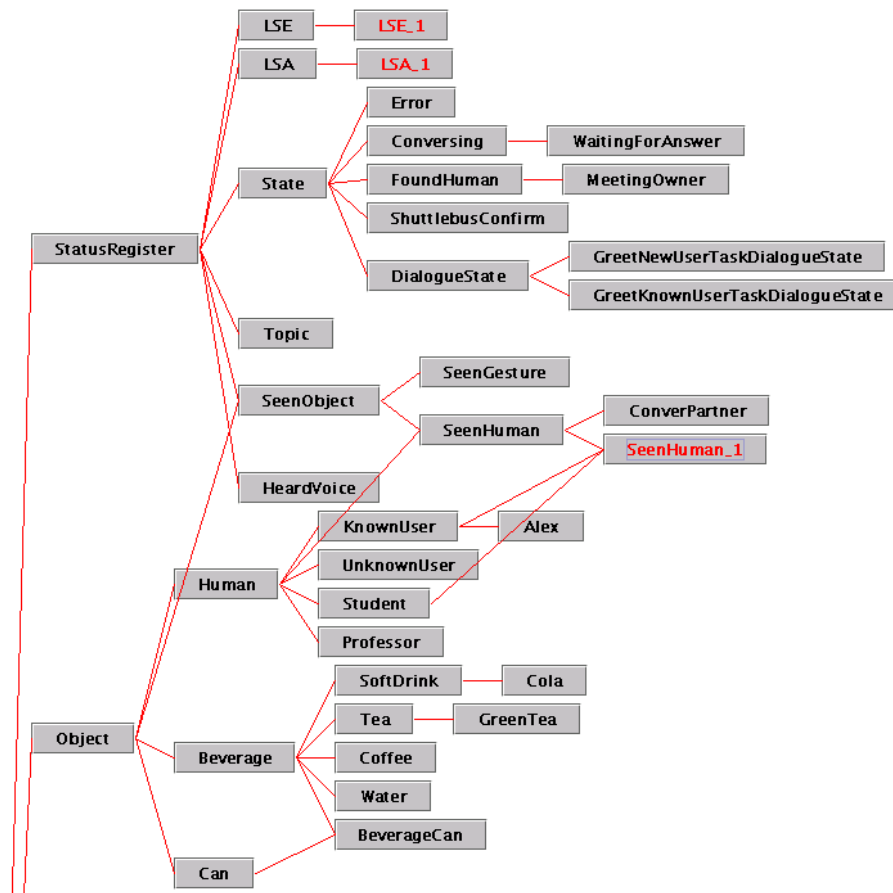
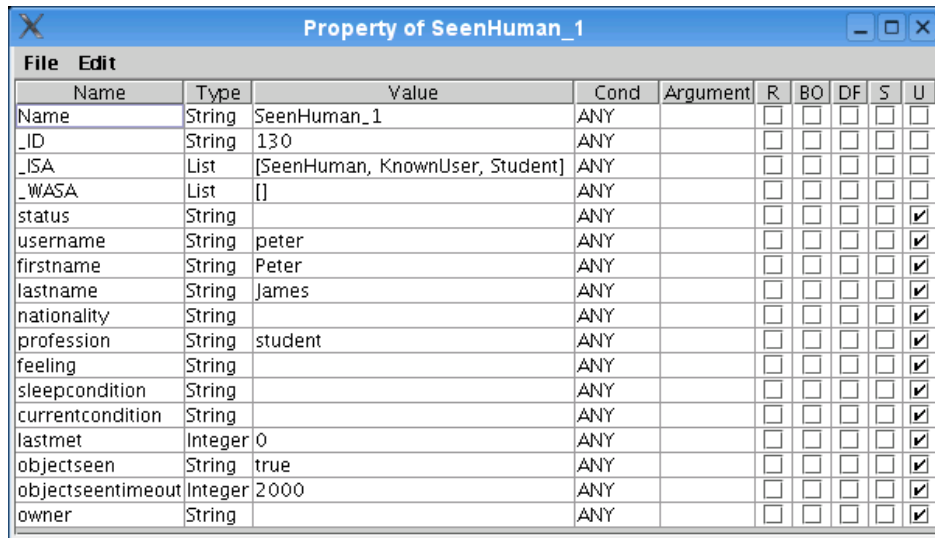


Figure 4.14: A knowledge hierarchy showing the *SeenHuman\_1* instance and its parents

Should there be any, SPAK will do so and execute the contents of special on-event slots *onInstantiate*, if it is not empty, which might cause outgoing actions via agents. That an instance is created, it can be **updated** both through direct commands from remote agents via network and GUI editor, or manipulation of the knowledge contents by Javascript procedures embedded in special slots. When it is updated, the *onUpdate* special slot is executed.

Over time, the *Evaluator* thread constantly **evaluates** each instance, executes its *onEvaluate* slot, checks its condition and induce and re-induces or deactivates the instance if needed. Modification of instance's parents, i.e., **add** or **remove parent (s)**, results in an execution of the *onTransition* and *onTransitioned* special slots. Finally an instance can be **deactivated** in which its *onDestroy* slot will be executed.

Based on the facilities provided by SPAK including these special slots and slot flags, one can design the knowledge frames hierarchies to represent the knowledge and manage robot actions. Our policy on designing the knowledge contents is to **follow the human's way of understanding the world of interest as much as possible but allow exceptions if there is a technical limitation**. For example, *Object* frames are used to represent tangible things like



Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	SeenHuman_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	130	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[SeenHuman, KnownUser, Student]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
status	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
username	String	peter	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
firstname	String	Peter	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
lastname	String	James	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
nationality	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
profession	String	student	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
feeling	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
sleepcondition	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
currentcondition	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
lastmet	Integer	0	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
objectseen	String	true	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
objectseentimeout	Integer	2000	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
owner	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

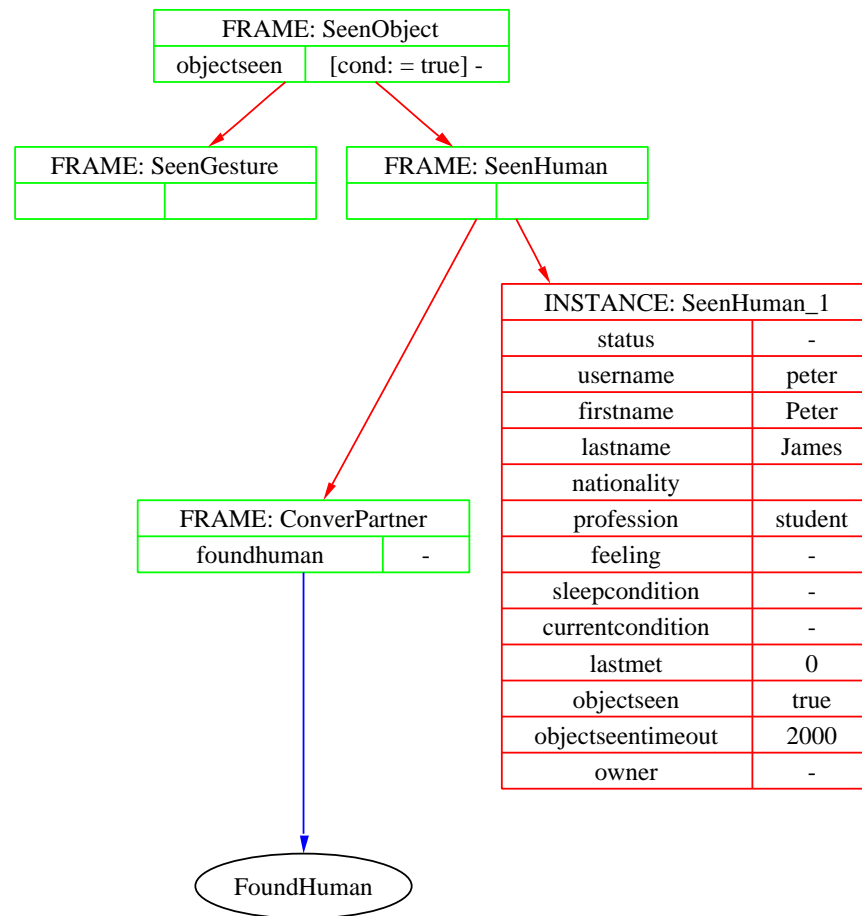
Figure 4.15: Property of the *SeenHuman\_1* instance

human, student, can, beverage. *Concept* frames are used to represent more abstract things like speech, gestures. As our use in robot systems involves much on management of robot event-action behaviors, therefore we have two more groups of frames: *Event* and *Action*. The event-action behavior is achieved by using the *Event* and *Action* frames. Event frames are designed for describing changes in the environment, e.g., *SpeechRecognized*, *FaceDetected*, *HeadSensorTouched*. When a primitive agent perceives a change in the environment, it passes the information to SPAK which will trigger instantiation of *Event* frames. It might further result in creation of new *Action* frame(s), e.g., *Say*, *Move*, causing action(s) in response.

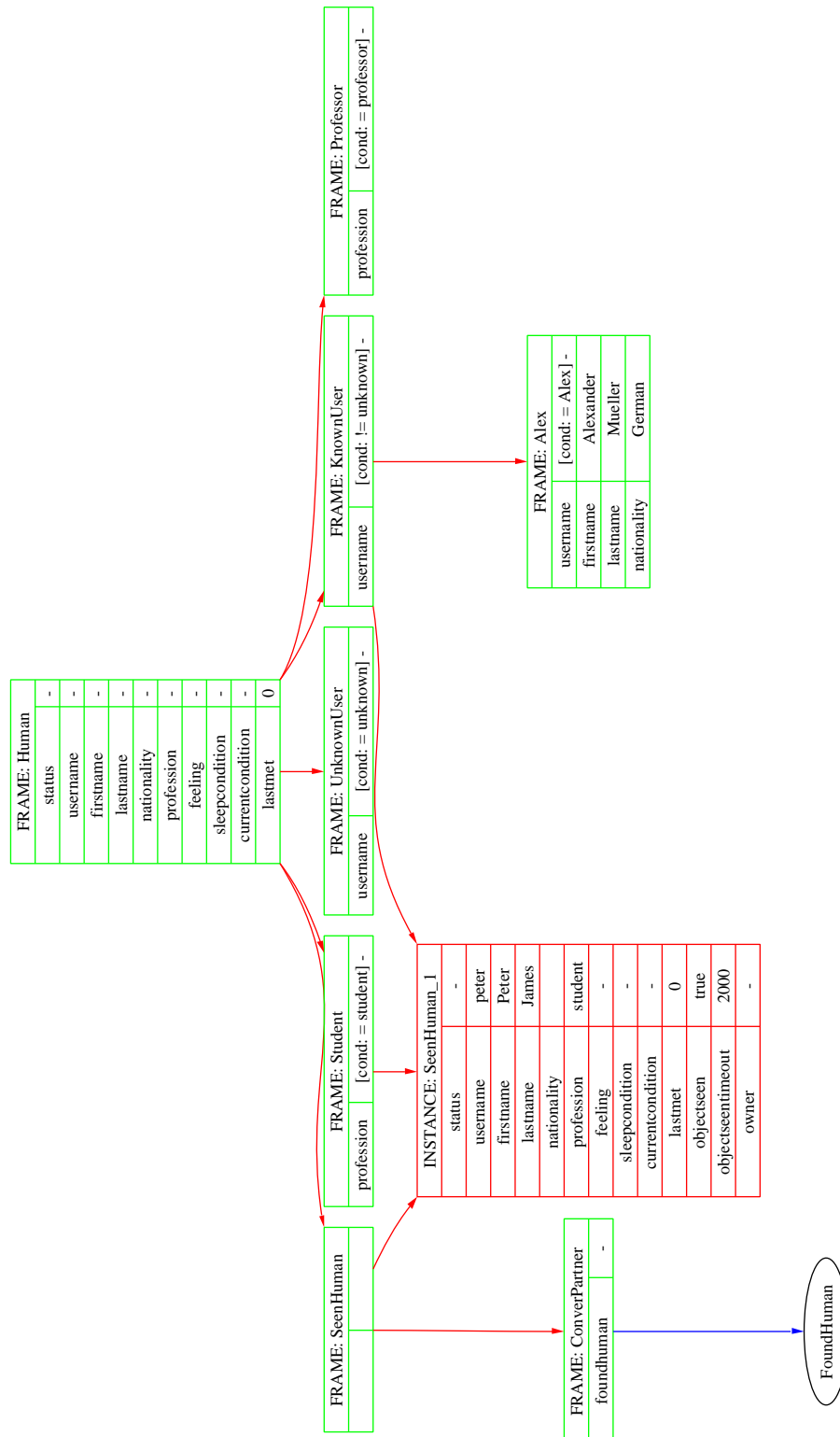
A sample used of these frames can be shown as follow. When an unknown face is recognized by a face recognizer agent, an event frame *UnknownFaceFound* is created in SPAK. Upon instantiation of this frame, the condition of an *Action* frame *Greet*, which requires an existence of an unknown person in front of the robot (i.e., the *UnknownFaceFound* frame) is fulfilled. Therefore a new instance of *Greet* frame is created and an action, e.g., say greeting word, is executed according to what specified in its *onInstantiate* slot.

Another design policy is that **multiple-parent frames are allowed and encouraged**, because things in the real world, as typically perceived by humans, are not in a single hierarchy. Therefore multiple-inheritance is encouraged. From the knowledge contents in Figure 4.4, one might notice that the *Line\_1* instance has two parent frames: *LongLine* and *ThickLine*, because its slot values match both frames' conditions.

Another example is that a human can have many roles, e.g., a human as a living creatures, a registered user to the system, and an object being seen by the robot. For example, a *ConverPartner* frame, which represents the current robot's conversation partner, is a child of a *SeenHuman* frame, with a required instance slot of type *FoundHuman* status register frame

Figure 4.16: A knowledge hierarchy starting from the *SeenObject* node

(more about status register frame will be discussed in the next chapter). The *SeenHuman* frame which represents a human being seen by the system is a child of the *Human* (representing a human as a living creature), *SeenObject* (object seen by the system) and *Student* (a human with the profession “student”) frames. Hence an instance of the *SeenHuman* frame can be a children of many frames at the same time. As shown in Figure 4.14, the *SeenHuman\_1* instance is a child of the *SeenHuman*, *KnownUser*, and *Student* frames. Figure 4.15 shows the slots and slot values of the *SeenHuman\_1* instance. The knowledge tree of its parent frames starting from the *SeenObject* and *Human* nodes are shown in Figure 4.16 and Figure 4.17 accordingly. The instance has slots’ values that match conditions of all its three parents (the *SeenObject* and *SeenHuman* frames require that the value of the slot *objectseen* is equal to true; the *KnownUser* frame requires that the value of slot *username* is not equal to unknown; and the *Student* frame requires the *profession* slot value to be “student”).

Figure 4.17: A knowledge hierarchy starting from the *Human* node

Method	Description
<b>Frame/Instance Related Methods</b>	
KFrame createFrame(fname)	create a new frame with the name <i>fname</i>
KFrame createInstance()	create an instance of this frame
KFrame findFrame(fname)	find a frame or instance which has the name <i>fname</i>
KFrameScript findFrame(fname)	find a frame or instance which has the name <i>fname</i>
Vector findInstancesOf(fname)	find instances of the frame whose name is <i>fname</i>
void selfDelete()	deactivate the current frame or instance
KFrameScript getKFrameScript()	get a JavaScript object that represents this frame or instance
Hashtable getHash()	get a hash table that represents this frame or instance
boolean tryMe(hash)	try if an instance of this frame can be induced by the provided information in the hash table
boolean isInstance()	check whether this object is a frame or an instance
boolean checkCondition()	evaluate if the condition of this frame is still valid
boolean isActive()	check if this frame or instance is still active
int getAge()	get the age of this frame or instance
int getAgeSinceLastUpdate()	get the age since the last modification to this frame or instance
<b>Parents/Children Related Methods</b>	
void addParentFrame(KFrame p)	add a new parent <i>p</i> to this frame
Vector getChildren()	get all children of this frame
void remove(c)	remove the frame <i>c</i> from my children list
void removeParent(p)	remove the frame <i>p</i> from my parents list
Vector getAllParentsFrames()	get all parent frames of this frame up the hierarchies (including grandparents, etc.)
boolean inParentsList(p)	check if the frame <i>p</i> is one of my parents
boolean inChildList(p)	check if the frame <i>p</i> is one of my children
void setToNewParent(np)	change my parent frame from the current one to <i>np</i>
<b>Slots Related Methods</b>	
void addSlot(slname, slval,...)	add a new slot to this frame with the name <i>slname</i> , value <i>slval</i> , ...
String getSlotValue(slname)	get the value of the slot <i>slname</i>
String getSlotValue(slname, timestamp)	get the value of the slot <i>slname</i> at time <i>timestamp</i>
String getPreviousSlotValue(slname)	get the previous value of the slot <i>slname</i>
void setSlotValue(slname, slvalue)	set the value of the slot <i>slname</i> to <i>slvalue</i>
boolean checkSlotsCondition()	check whether all slots' conditions are still valid
void runSpecialSlot(slname)	execute the JavaScript code specified in the special slot <i>slname</i>

Table 4.4: Some methods to manipulate frames provided by the *KFrame* Java class

Table 4.5: Some properties and methods of the *KFrameScript* class

Property	Description
<i>slotName</i> <i>SlotName()</i>	access the content of that particular non-procedural slot execute the JavaScript code specified in the procedural slot
Method	Description
<i>void die()</i>	execute the onDetroy special slot and deactivate this instance
<i>void init()</i>	execute the onInstantiate special slot

By mimicking the human way of understanding, the system can be easier made to understand symbols used by human compared to the design which focuses only on the functionalities. However, there are limitations in the expressiveness of the frame model, hence some exceptions are allowed, and work around to eliminate these limitations should be done. This policy allows practical knowledge design while still maintaining the similarities to the human perception of the real world.

## 4.7 SPAK Programming Interfaces

A knowledge frame in SPAK is technically a *KFrame* Java object. Table 4.4 lists important methods of the *KFrame* Java object. For the JavaScript code in the procedure-type slots or the input via the network gateway, both the *KFrame* class and the *KFrameScript* class can be used. *KFrame* is a native Java class exported to JavaScript environment. It is used for manipulate the frame structure, and the knowledge hierarchy. *KFrameScript* is a native JavaScript class representing the frame. It is used for manipulating the frame slot values inside JavaScript context. It provides conveniences, e.g., slot values can be accessed directly using the dot expression like `object.slotname`. Procedural slots can be called as `object.procedure()`. Table 4.5 lists some properties and methods of the *KFrameScript* class.

For example, to query and set the age of the *Child\_1* instance in the previous example, one can do like this:

```
$ telnet localhost 9900
Trying 127.0.0.1...
Connected to otto.
Escape character is '^]'.

$ s = Root.findFrameScript("Child_1")
Evaluate: s = Root.findFrameScript("Child_1")
Result is [Frame] Name=Child_1
Name=Child_1
_ID=9
_ISA=[Child]
_WASA=[]
age=14
[Frame] Name=Child_1
```



```

Name=Child_1
_ID=9
_ISA=[Child]
_WASA=[ ]
age=14

$ s.age
Evaluate:  s.age
Result is 14
14

$ s.age = 20
Evaluate:  s.age = 20
Result is 20
20

$ s.age
Evaluate:  s.age
Result is 20
20

$ s.getAgeSinceLastUpdate
Evaluate:  s.getAgeSinceLastUpdate
Result is 72
72

```

## 4.8 Summary

We introduced in this chapter the proposed extensions to the frame model and the implementation in our frame-based knowledge software platform SPAK. New extensions namely time-based layer, evaluator, and priority support are introduced to the frame model to represent changing knowledge and to support management robot behaviors by means of SPAK action mechanisms. SPAK is experimentally used in robot applications like dialogue manager (to be presented in the next chapter), scene understanding, tasks planning [10], and in ongoing work of gesture-based human robot interaction [108], and multi-robot collaboration [109].

SPAK is still an ongoing project. Latest information is available on the home page [110].

## Chapter 5

# SPAK Application: Knowledge-based Dialogue Manager

Now that we have SPAK as a platform for co-ordinating networked agents by means of the frame-based knowledge engine, in order for a robot to have some functionalities, a SPAK application shall be developed. Developing a SPAK application is basically to create the knowledge contents in SPAK, i.e., frame hierarchies, slot properties, and corresponding JavaScript procedures. This specifies how the system should behave, e.g., how to deal with the information received from agents and what actions should be sent to which robot agent.

This chapter presents the development of a sample SPAK application: a knowledge-based dialogue manager for robots. As our final goal is to achieve symbiotic robots that live together with and assist humans, especially the elderly and ill persons, considering these target users, it would be inconvenient to ask them to use the keyboard and mouse in order to interact with machines. Multi-modal natural interfaces like speech and gesture are more natural. Hence a dialogue system for robots is desired and it is chosen as a showcase for SPAK application development.

In this chapter we show the design of a knowledge-based dialogue system with the focus on the dialogue manager, the decision-making part of a dialogue system.

### 5.1 Design of the Dialogue System for Robots

An overview of the designed dialogue system is shown in Figure 5.1. The big dashed-rectangle represents the robot, which interacts with a human and the surrounding environment, perceives changes and generates actions in response. Inside the robot there are sensor-type and actuator-type agents, and SPAK, on which many robot applications are running, e.g., dialogue manager and gesture recognizer.

A basic task of a typical dialogue manager, as a part of a dialogue system, is to interpret

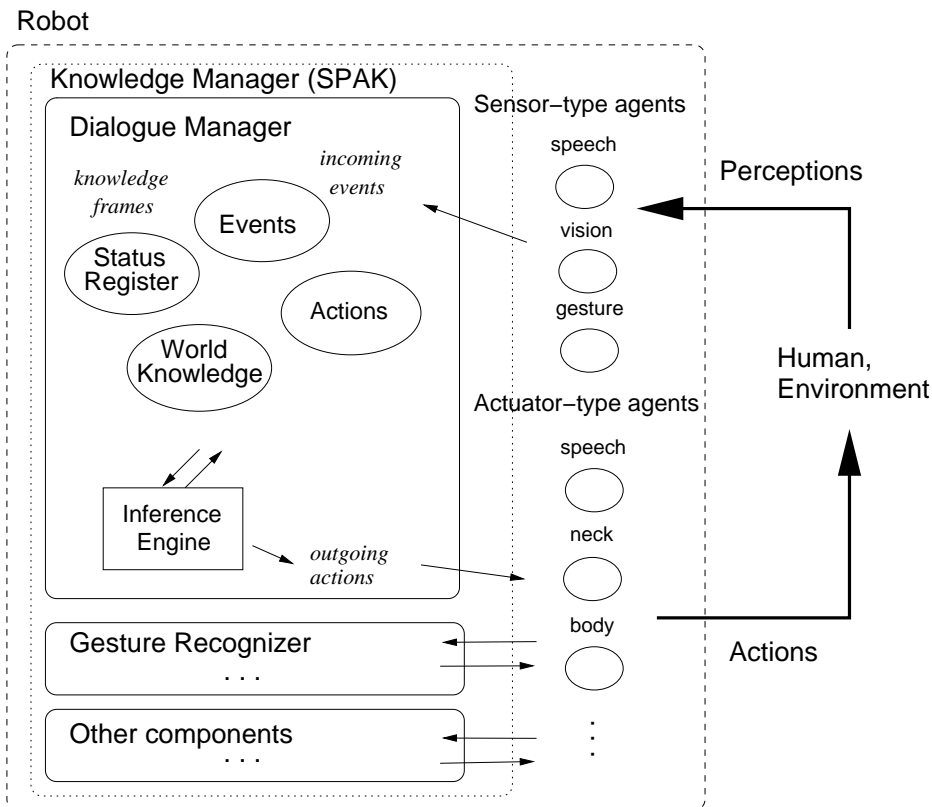


Figure 5.1: An overview of the designed dialogue manager. A robot is composed of various agents including the SPAK knowledge-manager agent.

input observations in context, and determine next action(s) to be done. In our robot case, the dialogs might compose of information queries, resource reservations or commands. Interactions are assumed not to be very complicated or ambiguous, considering our target users and that the human is aware of robot's limitation. On the other hand, the dialogue manager for robots must instead work well with other robotics devices and computing resources. Also as the robot is an integrated system with many applications running, not only the dialogue management but also other applications, e.g., movement planning, scene understanding; cooperation and sharing of knowledge among these applications are crucial.

Hence we propose a dialogue manager architecture based on the knowledge platform. Instead of having a dedicated system for each robot application, e.g., dialogue management, movement planning, scene understanding; a single knowledge platform serves as basis infrastructure for such applications. By sharing the single base platform, the system is less complex with single architecture; the dialogue manager can easily make use of the knowledge provided; and the knowledge can be easily shared among various applications.

The dialogue manager is targeted to understand human commands and be able to handle multi-modal state-based and form-based dialogue types, as they are the two most used

types of dialogues. It interacts with humans by means of various multi-modal sensory and actuator agents. Moreover, since the robot lives with us in the long term, it needs to maintain and use the knowledge about the world of interest which is changed during its life. An ability to update its behaviors through human instruction and feedback is also required because the robot cannot be pre-programmed with everything.

As the focus is on the dialogue manager, we made an assumption that speech input from users are sentences expected by the system, to isolate the dialogue management problems from other problems of natural language understanding, e.g., speech recognition and parsing problems. Sentences uttered by users will be recognized and parsed by responsible agents, and arrive in SPAK as an almost error-free communicative act information. In case of errors, the dialogue manager should inform the user to repeat the speech so that the dialogues can carry on.

## 5.2 Knowledge-based Dialogue Manager

A SPAK application is basically a collection of knowledge frames carefully designed to achieve certain robot behaviors. Knowledge frames that constitute the dialogue manager can be grouped as follows: *Events*, *Actions*, *StatusRegister*, and the world knowledge (including *Objects* and *Concepts*), as shown in Figure 5.2. Each group of frames is discussed in the following parts. In short, changes in the environment are captured by sensor-type primitive agents, forwarded to the SPAK and will be incorporated into the knowledge contents as *Event* frames. Adding this new information can result in modification of the *StatusRegister* frames, which include *state* frames indicating the current state(s) the robot is in, and/or other frames. From these changes, the inference engine might update other knowledge and/or generate *Action* frames, causing actuator-type agents to do some physical actions. Inference process can be triggered by the Evaluator process as well.

### Event and Actions Frames

Changes in the environment including the human actions, e.g. a speech is recognized, a face is detected, a face is recognized, or the user is pointing to the left, are represented in the knowledge base by *Event* frames as shown in Figure 5.3. *RawEvent* frames, whose children are, e.g., *SpeechRecognized*, *RawVisionEvent*, and *FaceDetected*, represent events frames that are created only from input information received directly from primitive agents. Other event frames are *WelfareCenterMessage* (simulating a message from a welfare center) and *GreetEvent* and *ByeEvent* (abstract frames representing actions of greeting from human which can be in speech, gestures, etc.).

The actions to be done by the robot are represented similarly using *Action* frames. In our design as shown in Figure 5.3, there are *BasicAction* frames for generating output actions

like *Say*, *Look*, *Pose*, and *Move*; *UpdateLSE*, *UpdateLSA*, and *UpdateSeenHuman* (more details about this in the following Status Register Frames subsection); *SpeechAction* for answering human requests via speech; *TaskAction* for handling multiple-step tasks including dialogue management (more details about this in Section 5.3); and *Learn* and *LearnedAction* for robot learning (more details about this in Sub-section 5.3.5).

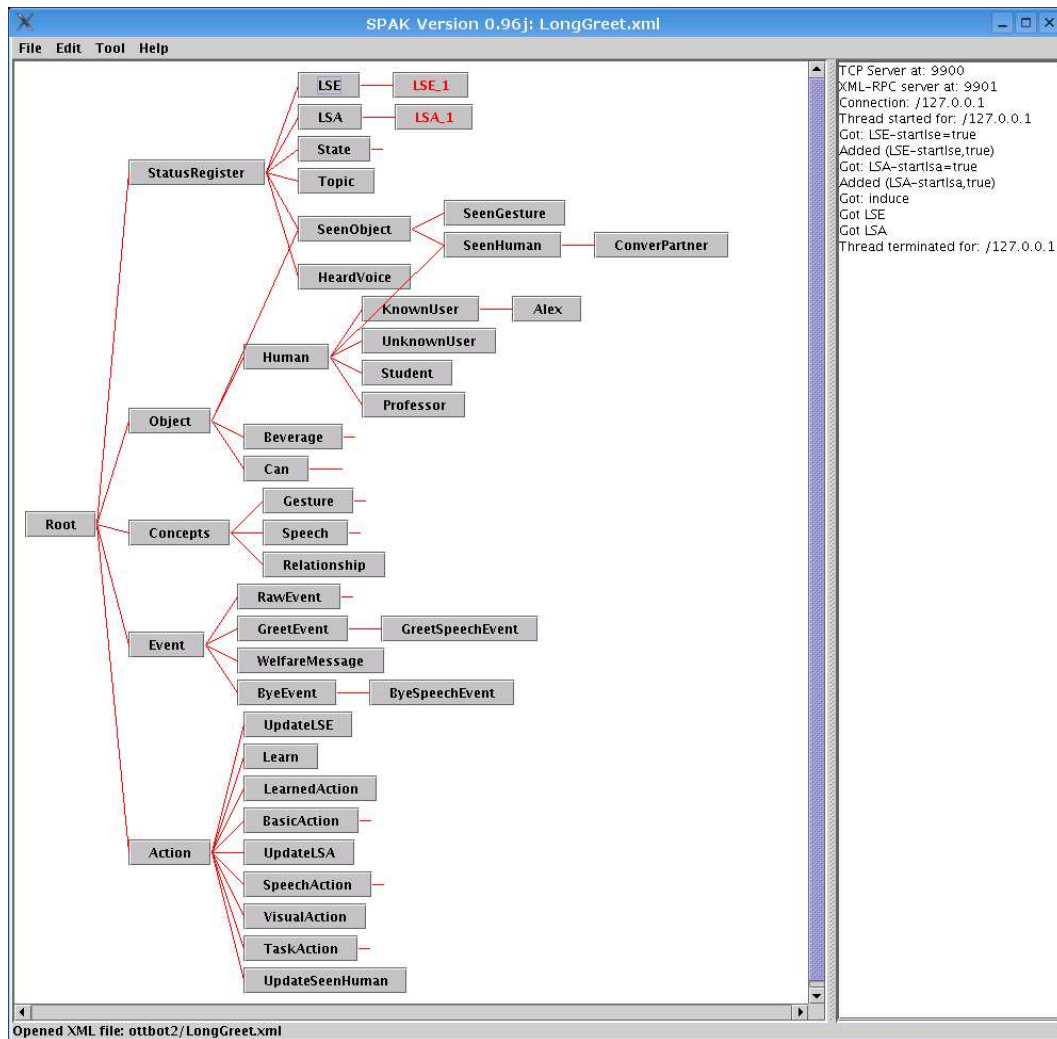
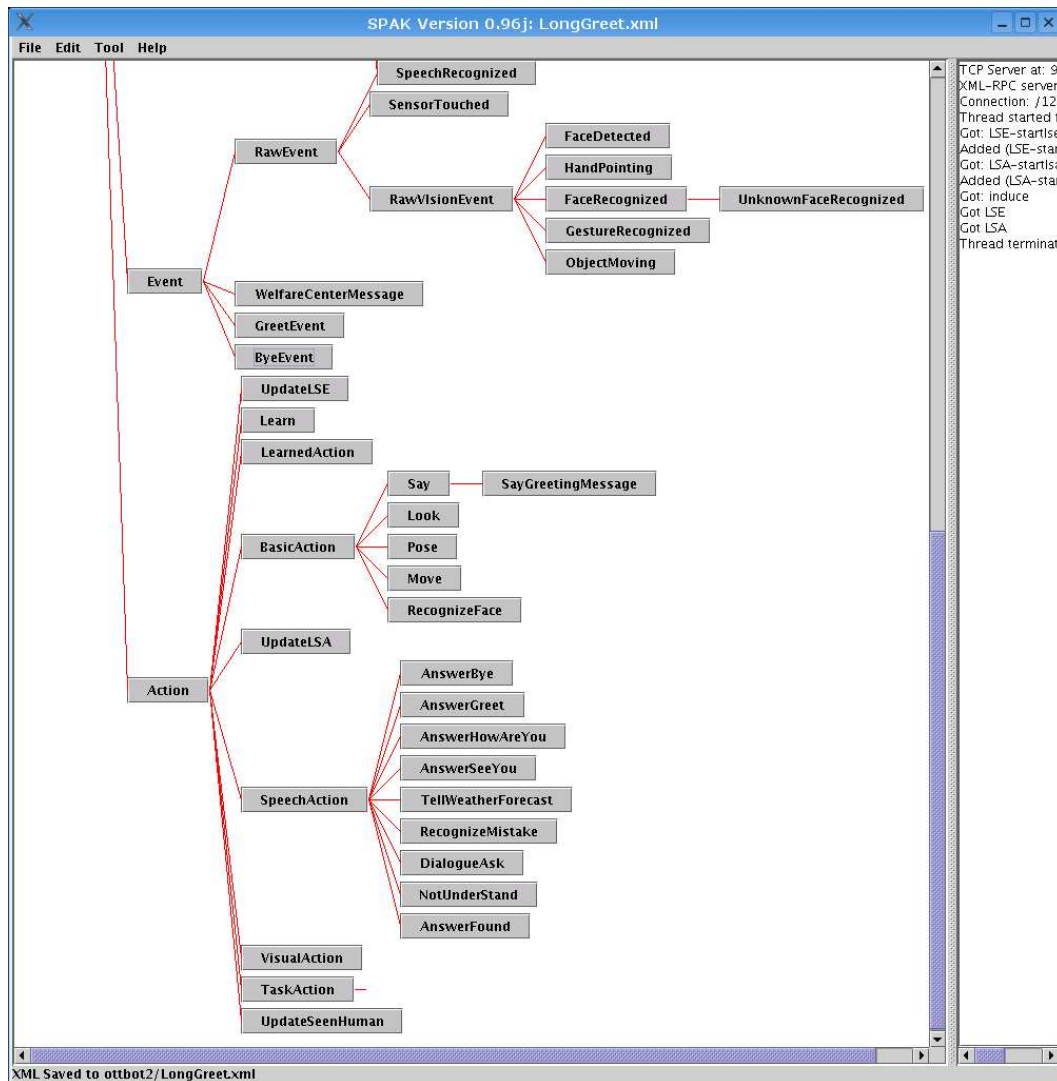


Figure 5.2: A screen-shot of SPAK showing groups of knowledge frames: *StatusRegister*, *Event*, *Action*, world knowledge (*Object* and *Concepts*), that constitute the dialogue manager for robots. Note that the frames further down the hierarchies are not shown.

### Status Register Frames

During a human-robot interaction, input data from other primitive agents come in continuously. Some actions can be triggered directly from this information, e.g., in case of human

Figure 5.3: A SPAK screenshot showing *Event* and *Action* frames

commands. However, some actions depend not only on a single event but rather on the current state or context. Status frames are designed to maintain the context information such as conversation partner, topic. Similar to the *Event* frames, they are instantiated when certain conditions are met. But they do not necessarily cause an action immediately. Instead, the frames exist and their values are updated over times to reflect the status of the system and the world of interest. This is realized using the mechanism of special slots and slot flags available in SPAK. Information provided by these status frames can be shared by other components of the system.

Status frames used in our demonstration scenarios are shown in Figure 5.4. Next we describe some of important status frames in our design.

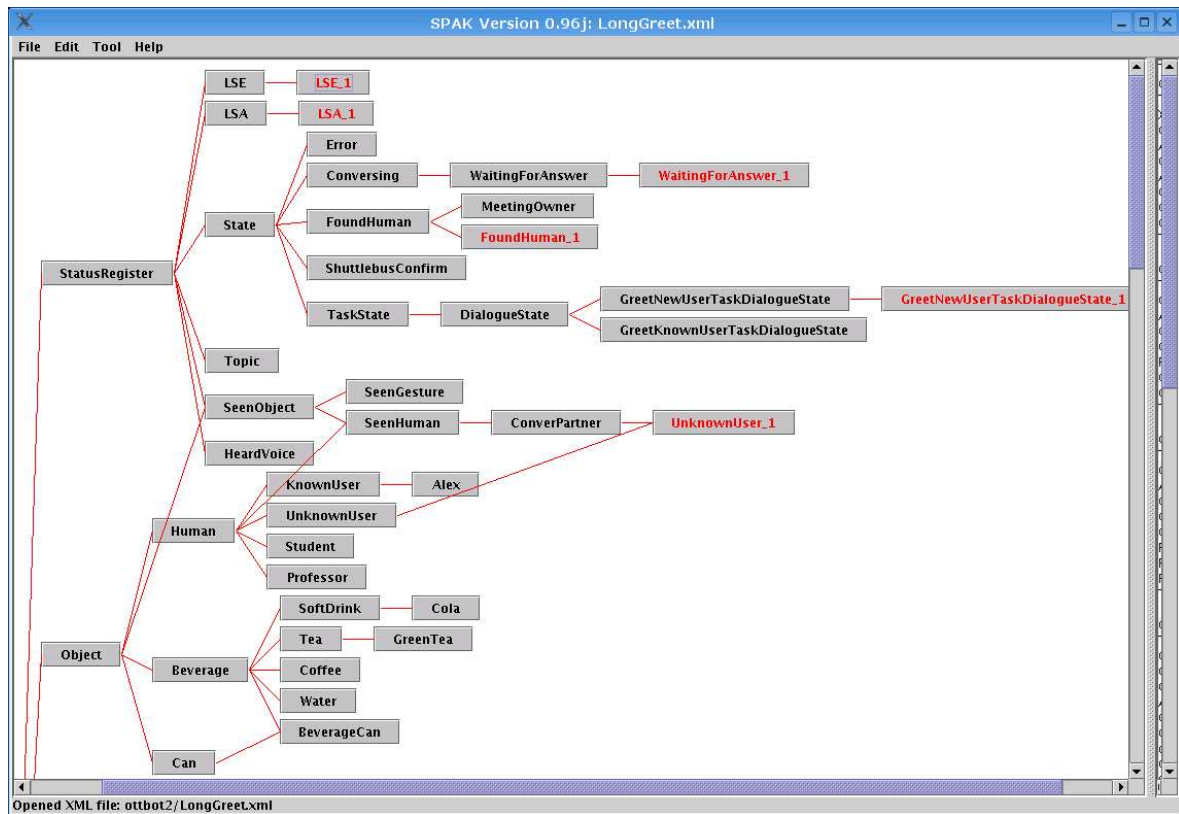


Figure 5.4: *StatusRegister* frames and *Objects* frames. Note some multiple-parent frames that are children of both the *Object* and *StatusRegister* frames

### LSE and LSA

During the interaction, many events and actions occur. In some cases the system needs to know what events have occurred or what actions it has done in the past. For example, in the case of learning from implicit instruction or feedback, the robot needs to know what is the thing human might be referring too. Two status frames: *List of Significant Events* (LSE) and *List of Significant Actions* (LSA) are designed to maintain the lists of the past events and actions accordingly, sorted by using the algorithm that hi-light rare and newer events. The action frames *UpdateLSE* and *UpdateLSA* are used to update the *LSE* and *LSA* accordingly.

Figure 5.5 shows the property of sample *LSE* and *LSA* frames. Top ten significant events are stored in the  $e1$  (most significant),  $e2$ ,  $e3$ , ...,  $e10$  (least significant) slots. Whenever there is a new *Event* frame instance, an *UpdateLSE* action will be created. It will update the *LSE* frame according to the policy that newer or rare events have higher priority. The algorithm to calculate this is as follows. Assume  $e_t^i$  is 1, if there is an event of type  $i$  occurred at the past time  $t$ , and 0 otherwise. The significance score  $s$  (value between 0 and 1) of the latest event

Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	LSE_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	125	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[LSE]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
priority	String	0	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
e1	String	UnknownFaceRecognized_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
e2	String	FaceDetected_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
e3	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
e4	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
e5	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	LSA_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	126	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[LSA]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
priority	String	0	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
a1	String	RecognizeFace_4	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
a2	String	Say_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
a3	String	RecognizeFace_3	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
a4	String	RecognizeFace_2	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
a5	String	RecognizeFace_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 5.5: A snapshot of the LSE and LSA frames

$e_0^i$  can be calculated using the window function  $w$  (see illustration in Figure 5.6) as follows:

$$s(e_0^i) = \frac{\sum_{j \in M} w(e_j^i)}{\sum_{t=1}^M e_t^i}, \quad w(e_t^i) = (a + (1 - a) \cos(\frac{t\pi}{M}))e_t^i$$

Using this calculation, newer events and rare events are given higher significance score. The parameters  $a$  and  $M$  can be adjusted to achieve the optimum result. Given a big enough size of the *LSE* and proper parameters, all the related events are likely to be found in the *LSE* list (this has yet to be verified). Updates of the *LSA* frame are done similarly to the *LSE*.

### States Frames

Other important status register frames are *State* frames. They are used to reflect the current state(s) of the system, e.g., *Error*, *FoundHuman*, and *WaitingForAnswer*. Some events or ac-



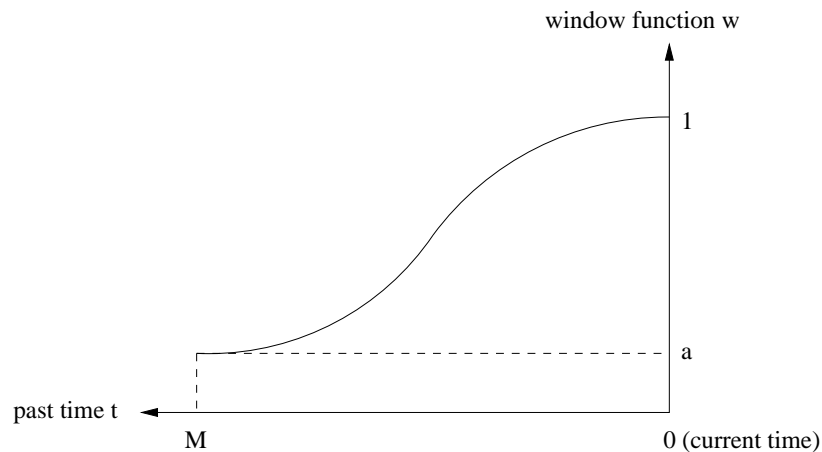


Figure 5.6: Window function used to calculate the significance factor

tions create or modify *State* frames. For example, while a human face is being detected, a corresponding *FoundHuman* state frame is created and maintained. After the human is being asked by the robot, a *WaitingForAnswer* state frame is created.

In the example screenshot of *StatusRegister* frames shown in Figure 5.4, the existence of the instances *FoundHuman\_1*, *WaitingForAnswer\_1*, *GreetNewUserTaskDialogueState\_1* indicates that a human is now present in front of the robot, the robot has asked a question to him and now is waiting for an answer, and the robot and human are interacting as a part of a so-called greet-new-user dialogue. The *UnknownUser\_1* corresponds to the human interacting with the robot. And that it is a child of *ConverPartner* and *UnknownUser* (and, up the hierarchies, *SeenObject*, *SeenHuman*, and *Human*) indicates that the human is being seen by the system, is the current conversation partner, and is unknown to the system.

This state information can be used by other parts of the system. Some actions can be done only if the system is in or not in certain state, e.g., not in the *Error* state. Note that there can be more than one state frames at a time, meaning that the system can be in different states at the same time. The significance of each state can be specified by in its *priority* special slot. For example, the *Error* state has the highest priority of 100, which means that even if the system is in several states at that time, the *Error* state overrides other states in case of rule conflicts.

## World Knowledge

The general knowledge about the world of interest, e.g., *Objects*, *Concepts*, is maintained in the world knowledge part. The world knowledge used in our dialogue system is shown in Figure 5.4 (*Object* frames) and Figure 5.7 (*Concepts* frames). *Objects* are used to represent tangible things like *Human* and *Can*, while *Concepts* are used to represent abstract things like *Gesture* and *Speech*.

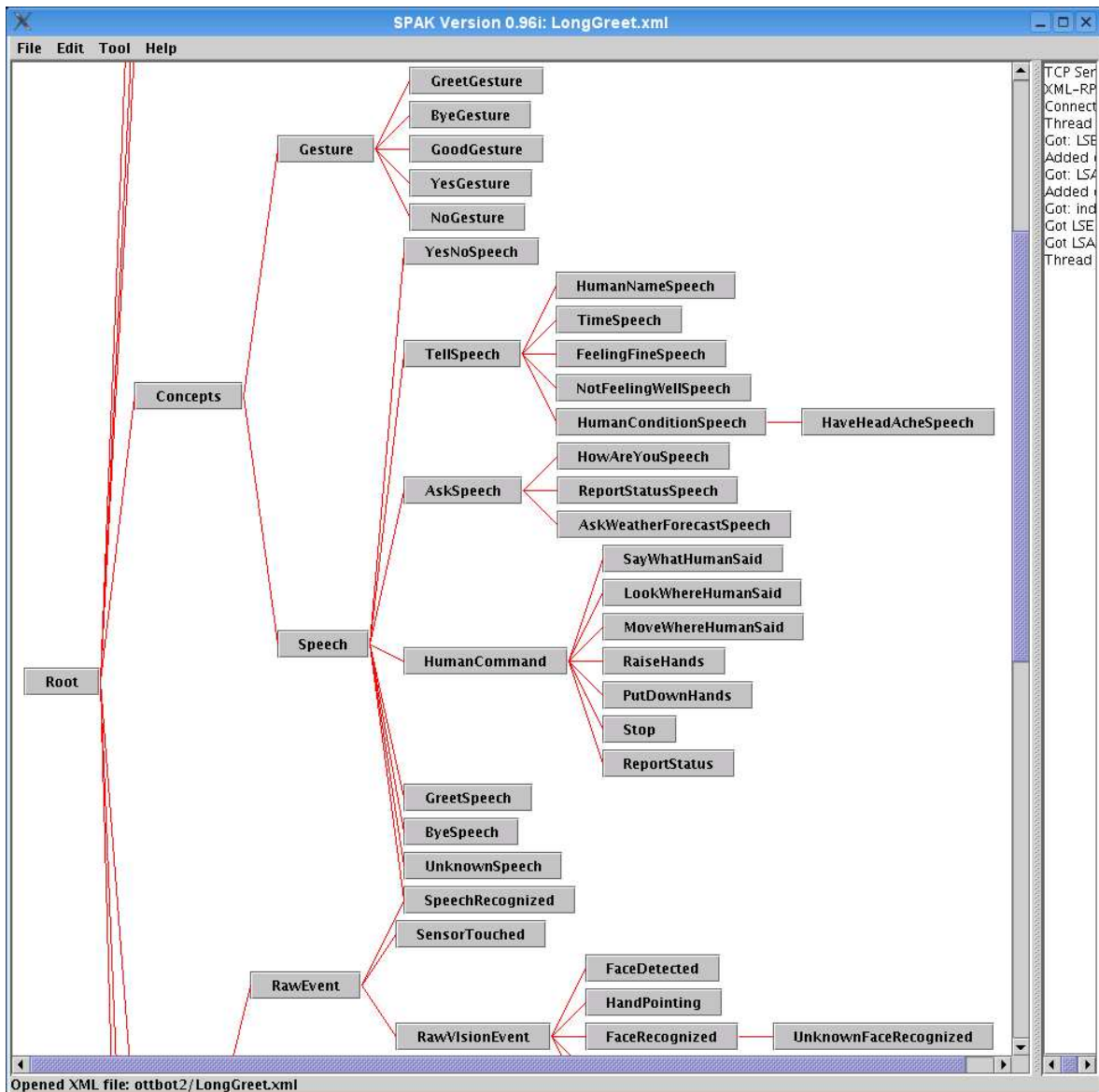


Figure 5.7: *Concepts* frames, as a part of the world knowledge, represent conceptual knowledge like gestures and speech

### 5.3 Dialogue Management

In this section, management of dialogue interactions by the designed dialogue manager is described. The discussion starts with the simplest form of interaction, handling of human commands, then proceeds to the method to ask a question, handling of state-based and form-based types of dialogues, and finally a basic robot learning from human instructions.

### 5.3.1 Handling of Human Commands

A basic interaction with the robot is to command it to do some actions. The robot can be pre-programmed to understand commands like say, move, look, and stop. In our design we use frames to achieve the command-action behavior. Related knowledge frames are shown in Figure 5.8. Commands like *SayWhatHumanSaid* and *LookWhereHumanSaid* are modelled as *HumanCommand* frames (which is a child of *Speech* and *Concepts* frame). In the bottom part of Figure 5.8, properties of the *SayWhatHumanSaid* frame is shown. By using this frame, when a human instructs “say <text>”, the robot follows the command by uttering that text. The *SayWhatHumanSaid* frame needs two instances of the *SpeechRecognized* event frames (see its *speech1* and *speech2* slots). Important is its *condition* slot, namely:

(*s.speech1.recognized\_text* == “Say” && *parseInt(s.speech1.getAge)* > *parseInt(s.speech2.getAge)*),

which means that if the first speech content is “Say”, this frame can be instantiated. Actions to be done when the frame is instantiated, e.g., utter the text as requested by the human (the text can be retrieved from the instance slot *speech2*, i.e., by calling *s.speech2.recognized\_text*) via a text-to-speech agent by creating a *Say* frame (a child of the *BasicAction* frame), can be specified as JavaScript code in its *onInstantiate* slot, or by using a separate *Action* frame that depends on this frame. Other *HumanCommand* frames are designed similarly.

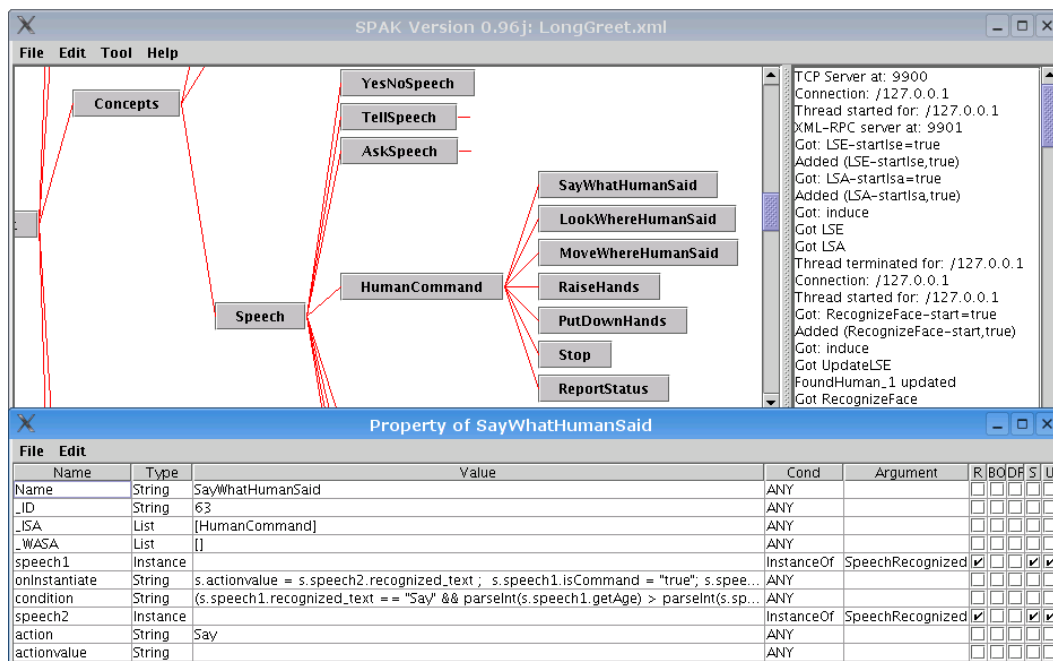
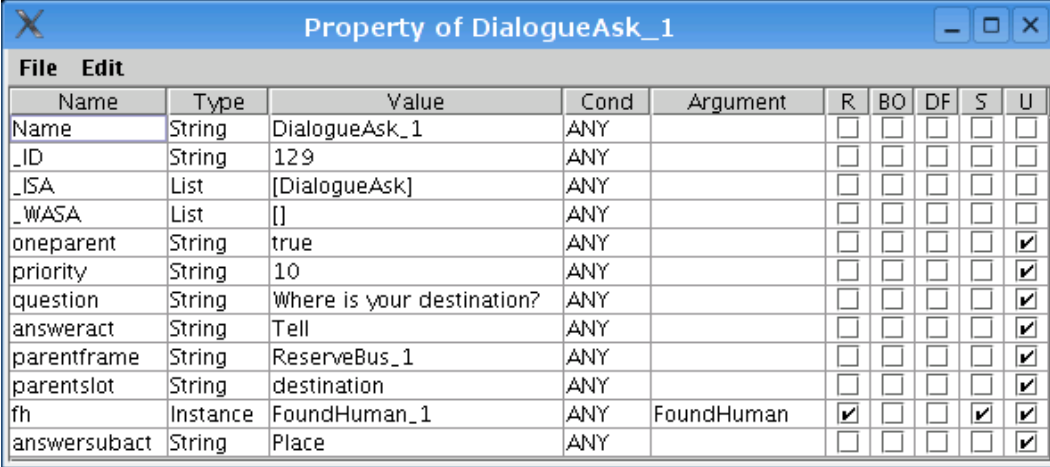


Figure 5.8: Example of the knowledge contents for processing robot commands, and the content of the *SayWhatHumanSaid* frame



Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	DialogueAsk_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	129	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[DialogueAsk]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
oneparent	String	true	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
priority	String	10	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
question	String	Where is your destination?	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
answeract	String	Tell	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
parentframe	String	ReserveBus_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
parentslot	String	destination	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
fh	Instance	FoundHuman_1	ANY	FoundHuman	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
answersubact	String	Place	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 5.9: A screenshot of a *DialogueAsk\_1* instance

### 5.3.2 Asking a Question

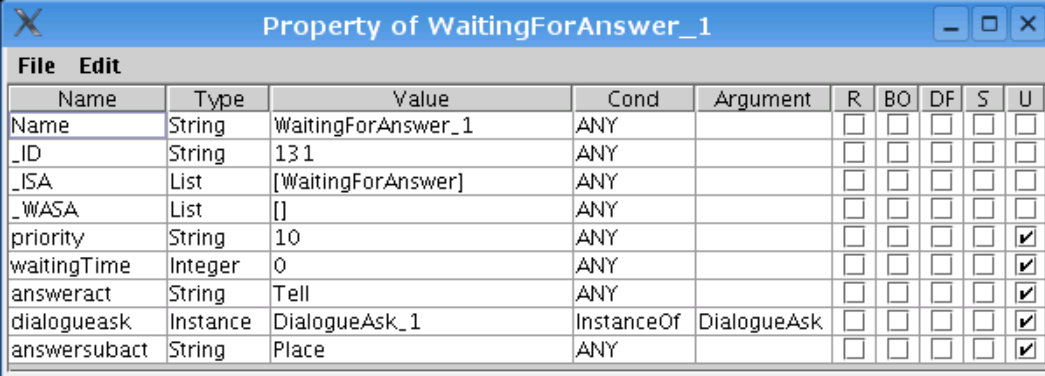
A number of frames are used to facilitate dialogue interaction with human, namely, the *DialogueAsk*, *WaitingForAnswer* and *AnswerFound* frames. When the system wants to ask a question to a human, a *DialogueAsk* action frame is created and filled with a question text and an expected type of answer. The property of a sample *DialogueAsk* instance is shown in Figure 5.9. It has important slots as follows:

- *question*: the question to be asked to the human, e.g., Where is your destination?
- *answeract* and *answersubact*: information about the communicative act that the system expects after asking the question, e.g., if the *answeract* is “Tell” and the *answersubact* is “Place”, the system expects that a human will say a place name.
- *parentframe* and *parentslot*: the parent frame and its slot name that the system should update with the information received in the answer. In the example in Figure 5.9, when a human reply is received, the system will update the *ReserveBus\_1* instance’s *destination* slot with the information about place it received.

When a *DialogueAsk* frame is instantiated, it creates a state frame *WaitingForAnswer*, which will ask the specified question to the human (by creating further a *Say* action frame). The existence of an *WaitingForAnswer* instance indicates that the system is now waiting for a response of the speech act type according to what specified in its *answeract* and *answersubact* slots. A sample *WaitingForAnswer* instance is shown in Figure 5.10. If the time passes without any reply, the system will ask the question again through the use of the *onEvaluate* slot in the *WaitingForAnswer* instance.

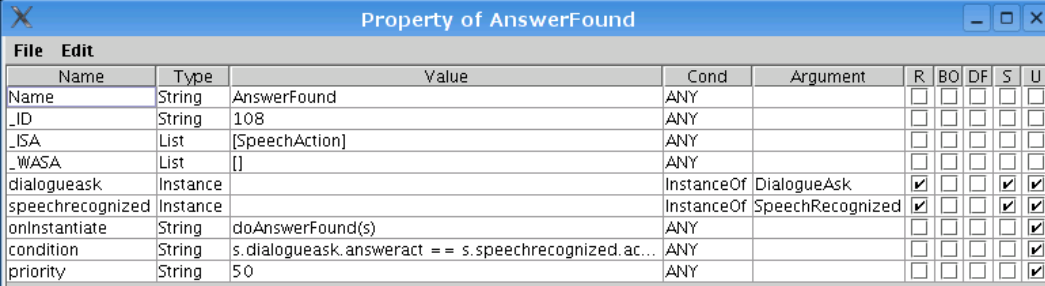
Once the human replied, a *SpeechRecognized* event frame is instantiated and an *AnswerFound* action frame, shown in Figure 5.11, will try to check if the *SpeechRecognized* event frame's speech act matches with the expected act specified in the *DialogueAsk* instance. If they match, the *AnswerFound* instance will update the frame specified in the *DialogueAsk* instance's *parentframe* slot with the answer from human. Finally, the task finishes, the *DialogueAsk\_1* and *WaitingForAnswer\_1* are deleted.

Related frames to the mechanism of asking a question and their interactions are illustrated in Figure 5.12.



Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	WaitingForAnswer_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	131	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[WaitingForAnswer]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
priority	String	10	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
waitingTime	Integer	0	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
answeract	String	Tell	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
dialogueask	Instance	DialogueAsk_1	InstanceOf	DialogueAsk	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
answersubact	String	Place	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 5.10: A screenshot of a *WaitingForAnswer\_1* instance created by the *DialogueAsk\_1* instance in Figure 5.9



Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	AnswerFound	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	108	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[SpeechAction]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
dialogueask	Instance		InstanceOf	DialogueAsk	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
speechrecognized	Instance		InstanceOf	SpeechRecognized	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
onInstantiate	String	doAnswerFound(s)	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
condition	String	s.dialogueask.answeract == s.speechrecognized.ac...	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
priority	String	50	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 5.11: A screenshot of an *AnswerFound* frame used to match between a *SpeechRecognized* instance representing incoming speech from human and a *DialogueAsk* instance

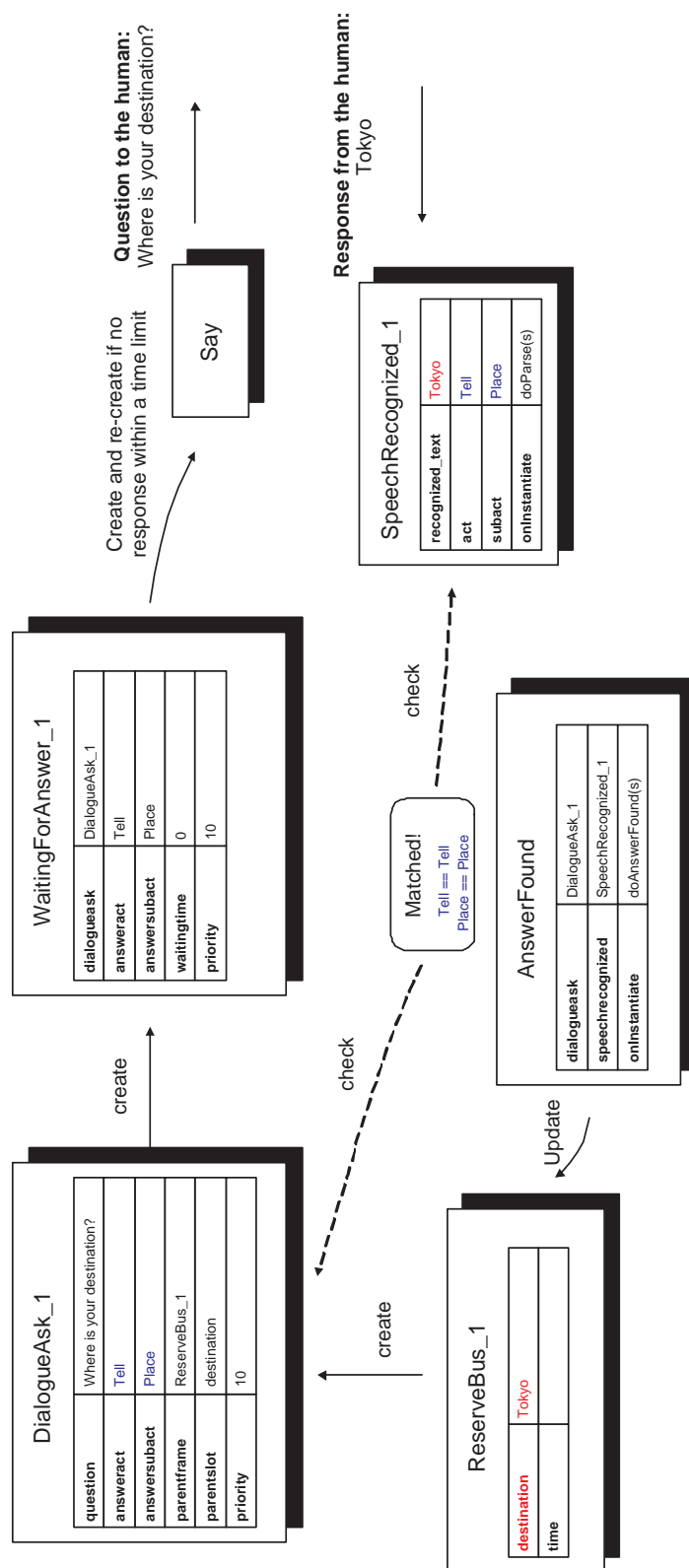


Figure 5.12: Related frames to the mechanism of asking a question and their interactions

### 5.3.3 Handling of State-based Dialogues

In this section we describe management of state-based dialogues. *DialogueTaskAction* frames, children of the Action frame, are used to manage the transition from one dialogue state to another. Figure 5.13 shows a sample set of the knowledge frames to realize state-based dialogues of greeting a new user (*GreetNewUserTaskAction* frames) and greeting a known user (*GreetKnownUserTaskAction* frames). Other frames involved are the *DialogueState* state frames (shown in Figure 5.4), which is used to maintain the current state and other shared information among the *DialogueTaskAction* frames.

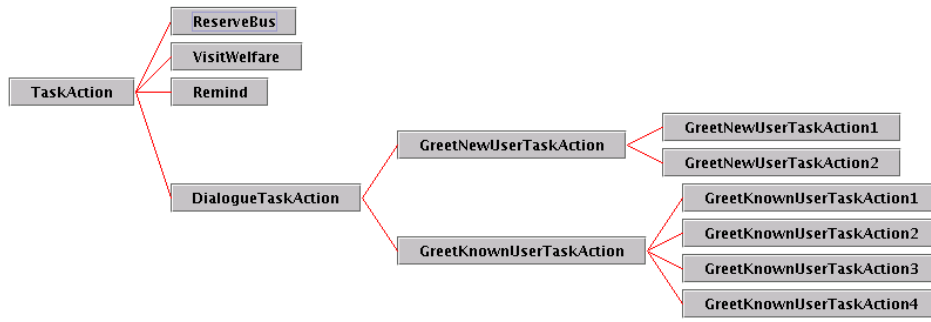


Figure 5.13: Example knowledge contents to realize state-based dialogues

Figure 5.14 contains a flow chart showing a task action flow, dialogue state frame updates, and interactions with human, in a state-based dialogue for greeting a known user (involving *GreetKnownUserTaskAction* frames and *GreetKnownUserTaskDialogueState* frames). The dialogue is started with some initial conditions, in this case, the conditions that a human is being in front of the robot (corresponding to an existence of a *FoundHuman* instance) and that the human is known to the system (corresponding to an existence of a *FaceRecognized* instance with a non-unknown human name). An instance of the first dialogue task frame *GreetKnownUserTaskAction1* is then created. It sends a greeting message to the human (in this example, “Hi Alex. How are you today?”), creates *GreetKnownUserTaskDialogueState* instance, and sets its *currentstate* slot to “howareyouasked”. Next, depending on a replying speech from the human, either a *GreetKnownUserTaskAction2* (if the human says “Fine, thanks”) or a *GreetKnownUserTaskAction3* (if the human says “No so fine”) frame is instantiated, which will generate a reply to human, and update the dialogue state frame accordingly. The dialogue progresses similarly until it reaches the *Finished* state.

### 5.3.4 Handling of Form-based Dialogues

The form-based dialogue, where a certain action is done once all the required information is provided, can be managed by using a *Task* frame. A small example of a scenario to reserve a

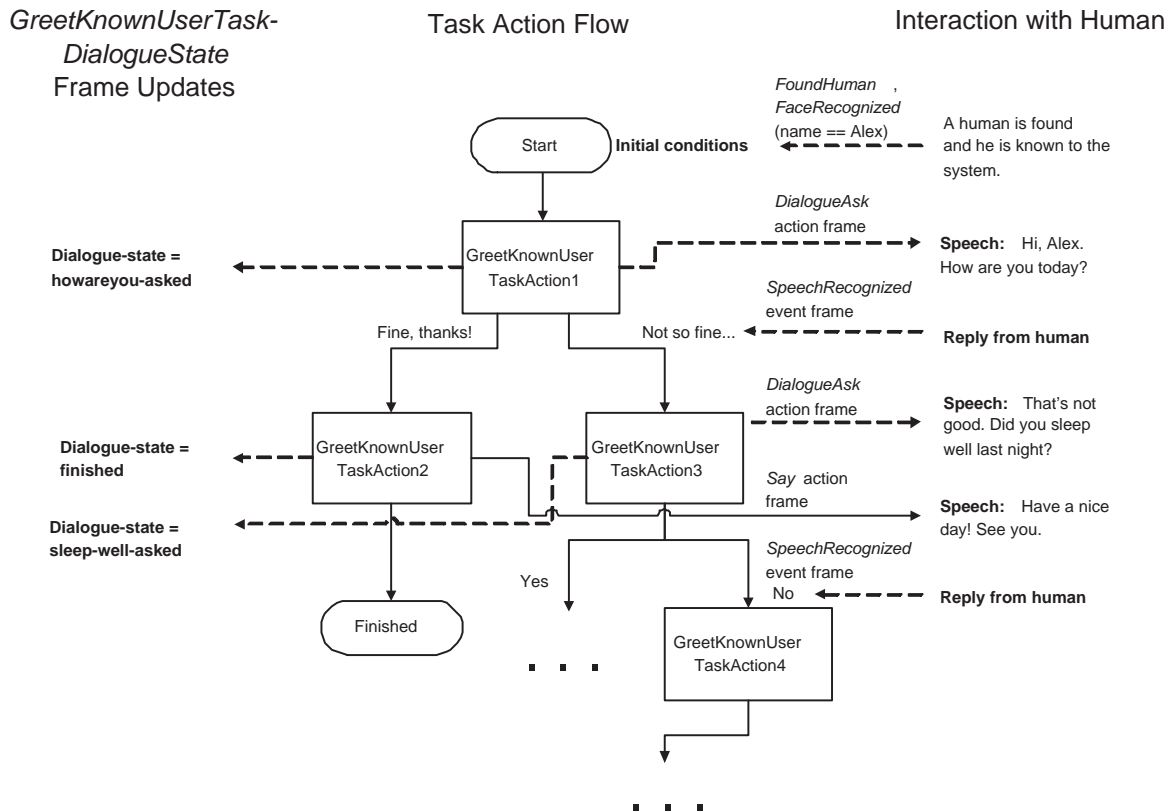


Figure 5.14: A flow chart showing a task action flow, dialogue state frame updates, and interactions with human, in a state-based dialogue for greeting a known user

bus seat is given. Involved frames are the *ReserveBus* task frame, and the *DialogueAsk*, *WaitingForAnswer*, *AnswerFound* frames, similar to in the state-based dialogue case. The steps are shown in Figure 5.15. The *ReserveBus* task frame, whose properties are shown in Figure 5.16, needs information stored in its two slots: *destination* and *time* in order to proceed with the reservation process. When it is instantiated, the script in its *onInstantiate* special slot, which calls the *onInstantiateReserveBus(s)* JavaScript function, will create two *DialogueAsk* instances: *DialogueAsk\_1* and *DialogueAsk\_2* instances, as shown in Figure 5.15, to ask the human these two pieces of information by means of the *WaitingForAnswer* and *Say* frames with the mechanism similar to what described in Section 5.3.2. However, a *WaitingForAnswer* frame will not ask the question if there already exists another *WaitingForAnswer* frame with a higher *priority* value. Instead, it will wait until that question has got the answer, i.e., until that higher-priority *WaitingForAnswer* instance disappears. This mechanism prevents two (or more) *WaitingForAnswer* instances from interfering with each other. If the user provides both information at once (i.e., the system asks for time but the human provides both the time and destination information), the system will not ask the rest of the question since



it has the answer already.

Once the user replied, his speech is represented by a *SpeechRecognized* event frame. During its instantiation process, the speech text is parsed and the resulting performative act is stored in the slot value *act*. Again, the action frame *AnswerFound* is used to match this with the existing *WaitingForAnswer* instance's. If they match, i.e., their speech *acts* and *subacts* are the same, the *AnswerFound* instance will update the *ReserveBus* frame's corresponding slot (time or destination) by checking the *parentframe* and *parentslot* of the *DialogueAsk* instance that created the *WaitingForAnswer* instance. Then it deactivates the *DialogueAsk* frame instance. Once all the required information in the *ReserveBus* frame are filled, the reservation process can be started. This checking and finalizing are done by the function *onEvaluateReserveBus(s)* in the *onEvaluate* slot of the *ReserveBus* frame.

### 5.3.5 Robot Learning from Human Instruction

Since our target application is a welfare robot which will live with us in long term, not only should the robot be able to communicate with humans in pre-defined dialogues, but also to learn through the interaction in order to improve itself to serve human better. In this research we are concerned with the robot learning of new facts and rules through the conversation. Learning can be done in various ways, for instance: learning from explicit human instructions (e.g., the human teaches "Alex is a man"); learning from implicit human instructions (e.g., human points to the right and then teaches the robot to turn to that direction; the robot has to find out why it should turn to that direction by looking at the previous incoming events and pick up the most promising one); reinforcement learning, and learning by demonstration.

In the first stage we focus on learning by explicit and implicit human instructions. In the explicit case, it is simpler; once the system understands human's teaching messages, it can simply add the new knowledge into its knowledge base. For example, when the human says "Alex is a man", the robot should add an *IS-A* relationship between *Alex* and *Man* (it is assumed that the robot knows what *Human* is). For the sentence "if you meet Alex, tell him the books have arrived", the robot should learn that a *Tell* action with message content "the books have arrived" should be executed when it meets *Alex*. In such cases, the parser needs to understand the input text and deliver the correct communicative act. After that the knowledge server should process it and take appropriate actions.

In the implicit human instruction case, the robot has to observe input events from the environment including human's teaching messages, and tries to find what the human intends to teach. For simplicity, we started with the learning of rules that has only one single triggering event, e.g., if the human greets the robot with words "Hello", it should respond with "Hello". The question is, among a lot of incoming input events, how the system knows which is the right information, e.g., the human's Hello message in this case.

In the second demonstration scenario in the next section, we show a simple robot learning from implicit human instructions. There are two *Action* frames involved in the learning: *Learn* and *LearnedAction*. The *Learn* frame takes care of the learning process. It is created when human issues a command. Then it tries to find an event from *LSE* (at the moment it simply takes the top of the list, i.e., the *e1* slot) and associates with the command by creating a new *LearnedAction* frame with the starting *confidence* slot value of 1 (means one teaching sample). For example, human comes to the robot, let his face detected by the robot (i.e., there will be a *FaceDetected* event frame) and asks the robot to say “Hello” (i.e., the *SayWhatHumanSaid* human command frame will be created). The robot should associate that *FaceDetected* event frame with the action saying “Hello” and create a corresponding *LearnedAction* frame. This new *LearnedAction* frame will require an *Event* frame of type *FaceDetected* and the action when it is instantiated is to say the text “Hello”. The robot has learned a new behavior from human instruction.

Human feedback messages like “Good” or “No, don’t do that” can be made to increase or decrease the value of the *LearnedAction*’s *confidence* slot. We can set a threshold for this *confidence* value to control whether this *LearnedAction* should be enabled or not. Thus by giving a positive or negative feedback, robot behaviors can be changed accordingly.

This is simply to show that the knowledge contents that controls robot behaviors can be modified on the fly during the interaction. We are by no means proposing a new learning algorithm. However, robot learning, especially how to make the robot learn from human instructions to improve its service, is considered one of our future topics.

## 5.4 Summary

We proposed the concept and implementation of a dialogue manager that is completely based on the extended frame-based knowledge platform. The designed dialogue manager can handle multi-modal state-based and form-based types of dialogues, and following human commands. It can also do basic learning from human instructions.

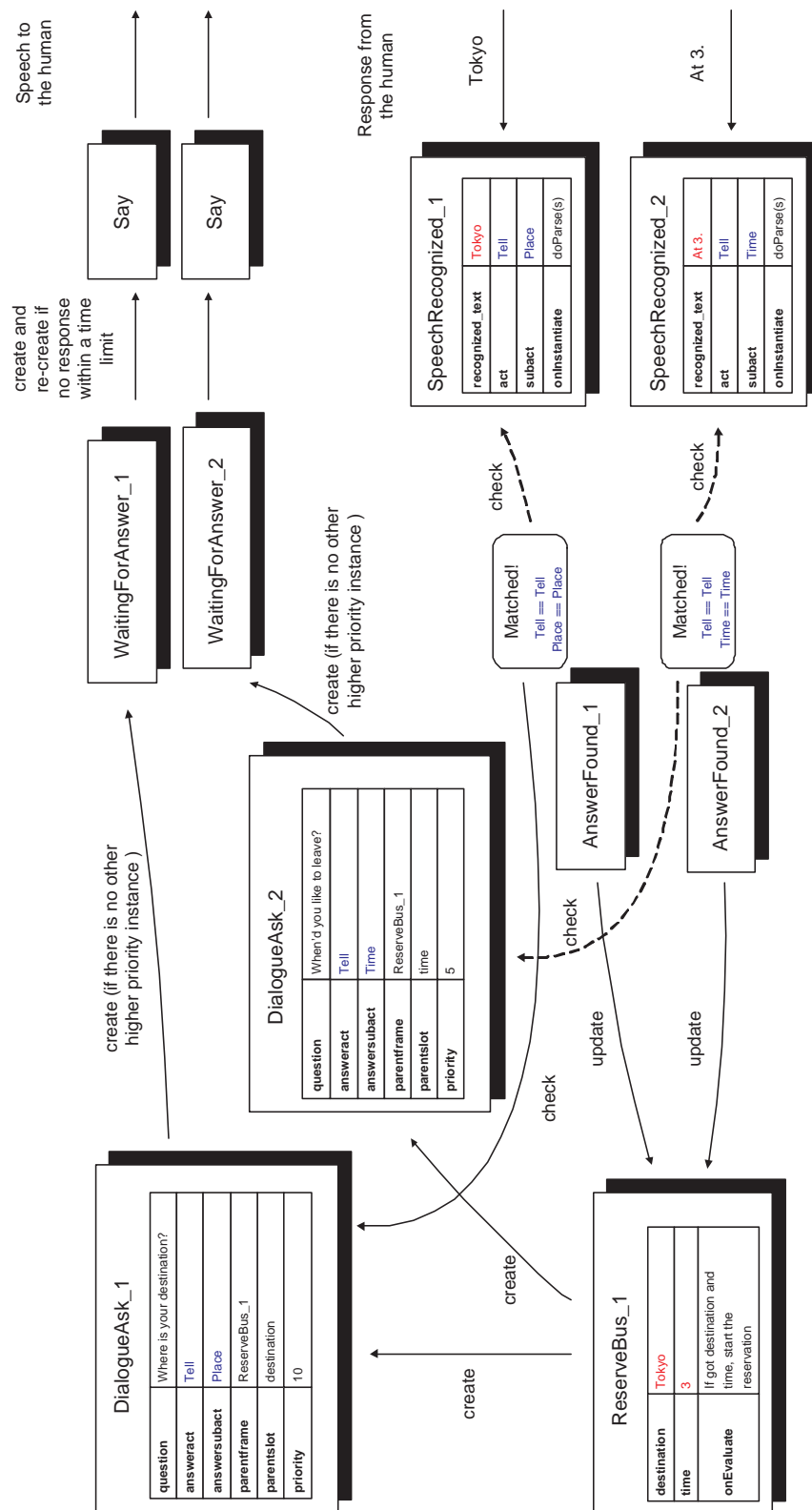
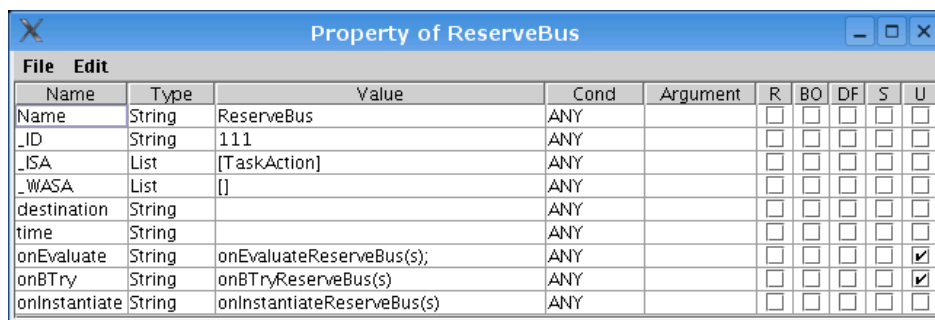


Figure 5.15: Related frames to the bus reservation dialogue and their interactions



Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	ReserveBus	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	111	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[TaskAction]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
destination	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
time	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
onEvaluate	String	onEvaluateReserveBus(s);	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
onBTry	String	onBTryReserveBus(s)	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
onInstantiate	String	onInstantiateReserveBus(s)	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 5.16: Property of a *ReserveBus* frame, which is an example of a frame-based dialogue with the goal to reserve a bus seat

## Chapter 6

# Prototype Development

In this Chapter, the technical design and implementation of the prototype system is discussed. The robot is composed of multiple networked agents according to the design in Chapter 3. The knowledge platform SPAK discussed in Chapter 4 serves as a knowledge manager agent on the network. On top of SPAK, a dialogue manager application discussed in Chapter 5 is running.

First we discuss the robot's components, i.e., the agents that compose the robot. Then we go on to the knowledge contents design in SPAK. Finally we show the details of three human-robot interaction scenarios and describe how the system works internally.

### 6.1 Robot Components

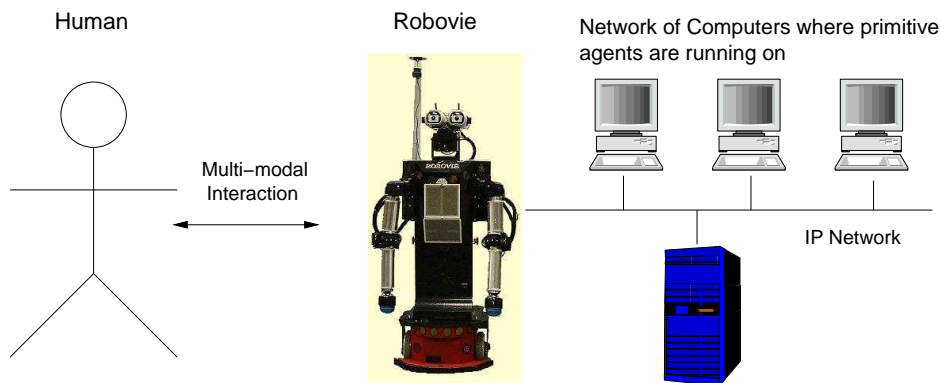


Figure 6.1: The prototype system

The Robovie-II humanoid robot [35] is used in the prototype system. It is developed by ATR, Japan. Robovie has a human-like upper torso and an ActivMedia wheels base. In this setting, Robovie interacts with human using speech, vision, and gesture, by moving its arms

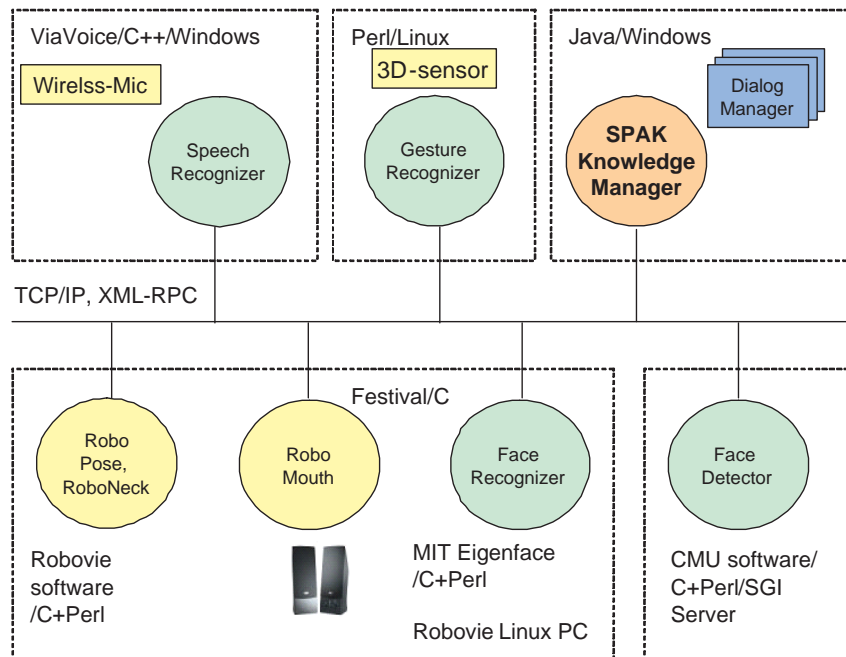


Figure 6.2: System configuration of the prototype system. Agents in the demonstration prototype run on five different machines and communicate on a TCP/IP network using the XML-RPC protocol.

and head. It has two eye-cameras and a speaker at its mouth. As it was not equipped with microphone, a wireless microphone is attached to the human's head instead. Movement control software is installed on the Robovie's internal Linux PC.

Apart from Robovie, there are networked computers running SPAK and other agents. An overview diagram of the system is illustrated in Figure 6.1. The system is composed primitive agents, namely, *RobovieNeck*, *RoboPose*, *FaceDetector*, *FaceRecognizer*, *SpeechRecognizer*, *RobovieMouth*, *GestureRecognizer*, and *Knowledge Server*, running on different networked computers.

System configuration of the prototype is shown in Figure 6.2. Details of each agent, including important functions, are discussed as follows.

## FaceDetector

The face detection software developed at the Carnegie-Mellon University [111] is used to find face locations in images in the FaceDetector agent. The agent receives input images via the *setImage()* function. Locations of the detected faces can be retrieved by calling the *getFaceLocations()*. The agent can be also set to automatically submit a reporting frame to the

Knowledge server whenever a face has been found in the image.

The access interface of the FaceDetector agent is shown in Chapter 3. In typical uses, first SPAK calls the *setSPAKIP()* function to inform the agent the IP address of the knowledge server. A Perl program is used to regularly access a Robovie video camera, take snapshot images of size 320x240 pixel (at the moment only 1 eye camera is used) and feed them to the face detector agent. When a face is found (a face is considered found when it is continuously detected two times consecutively), the agent submits the information “*FaceDetected-status=present*” to SPAK at the provided IP address. When the face disappears (similarly, this means the result of face not found two times consecutively), it sends the message “*FaceDetected-status=absent*” to SPAK.

### FaceRecognizer

This agent performs face recognition using the Eigenface software developed at MIT [112]. FaceRecognizer will not do the face detection, therefore input images are required to contain only the face area. The recognition result can be either names of the known persons, or unknown. Its interface includes following functions:

- *void resetDB()*: clear the database.
- *string recognize(base64\_encoded\_data imagecontents)*: recognize the face in the *imagecontents* image data.
- *void setName(string name)*: assign *name* to the last recognized-as-unknown person and add to the face database.

### SpeechRecognizer

The IBM ViaVoice speech recognition software (command menu mode, limited English vocabulary) is used as back-end in the SpeechRecognizer agent. The agent runs on a PC which is connected to the wireless microphone worn by the human subject. It has the following interfaces:

- *void setSPAKIP(string ip\_address)*: inform the knowledge server at the specified IP address if a speech is recognized.
- *string recognize(base64\_encoded\_data soundcontents)*: recognize the speech given in the *soundcontents* parameter.

Two mode of usages are possible. If the function *setSPAKIP()* is called, it enters the automatic mode. In this mode, the agent keeps endlessly processing the sound input from the microphone. As soon as a speech text is recognized, it sends an information “*SpeechRecognized-text=<text\_string>*” to SPAK. In another mode, i.e., the manual mode, it waits for calls to

the *recognize()* function in which it can get input speech in either compressed Ogg Vorbis format or uncompressed WAVE format. After that it processes the speech and returns the recognized text string as output.

Note that for the text parsing task, a SPAK-based simple pattern-matching parser written in JavaScript is used, no separate parser agent is running.

### RobovieMouth

RobovieMouth is connected to the sound device and speakers of Robovie. It accepts strings of input text and forwards to the Festival Text-to-Speech software [113], which will generate speech from the text, and output to Robovie's speakers. RobovieMouth also supports raw sound input contents. It has the following interface:

- *void sayText(string text)*: output the speech according to the given text string.
- *void utter (base64 soundcontent)*: output the given sound contents

### GestureRecognizer

This agent can detect only pointing directions: pointing left, right, and center, using the Polhemus 3D magnetic sensor. It has the following interface:

- *void setSPAKIP(string ip\_address)*: inform the knowledge server at the specified IP address if a gesture is recognized.

In typical uses, the agent is first given an IP address of the SPAK knowledge server via its *setSPAKIP()* function. When it detects a new gesture, it submits a corresponding information to SPAK, e.g., "*GestureRecognized-act=Pointing\nGestureRecognized-subact=left*".

### RoboPose

RoboPose offers low level functions to control the movement of Robovie's mechanical parts. Its interface is as follows:

- *void playPose(string pose\_string)*: instruct Robovie to pose according to the specified pose in the form of position values for its 13 joints.
- *void goZeroPosition()*: reset the Robovie pose to the zero or starting position. Basically this method will call *playPose("0 -5 0 -5 0 -5 0 -5 0 0 0 0")* which gives the *playPose()* function the zero-position values for all joints.

Apart from these two functions, more functions are provided to instruct Robovie to make pre-defined robot poses. These functions call *playPose()* similar to the *goZeroPosition()* function but with different pose strings. For example,



- *void byebye()*: perform a bye-bye gesture, which will technically submit three pose strings to *playPose()* to make a waving-hand farewell gesture:
  - "+2.182540 +1.845238 -0.338889 +0.119048 +0.000000 -5.000000 +0.000000 -5.000000 +0.000000 +0.000000 +0.000000 0 0";
  - "+1.706349 +2.321429 -0.048413 +2.460317 +0.000000 -5.000000 +0.000000 -5.000000 +0.000000 +0.000000 +0.000000 0 0";
  - "+1.825397 +2.678571 -0.242063 -1.031746 +0.000000 -5.000000 +0.000000 -5.000000 +0.000000 +0.000000 +0.000000 0 0";
- *void pointleft()*: raise the left hand and point to the left.
- *void pointright()*: raise the right hand and point to the right.

## RobovieNeck

RobovieNeck offers higher level functions to control the movement of the robot's neck (e.g., move left, move right, and go to the zero position). For simplicity, the destination is in the form of (x,y) coordinate, each with the value ranging from -2 to 2. Therefore there are total 25 possibilities of positions. Important functions offered by RobovieNeck are as follows:

- *void goZeroPosition()*: reset the neck position to the starting position.
- *void moveRel(string direction, int distance)*: move the neck in the given *direction* for the given *distance* relative to the current position.
- *void move(int posX, int posY)*: move to an absolute position at the coordinate of (posX, posY).
- *void followObject(int posX, posY)*: move the neck so that it follows the object at the position (posX, posY).

## RoboSensor

RoboSensor monitors all the tactile sensors on Robovie's body, e.g., head, arm, belly, and chest, and contacts SPAK if a sensor is touched by a human. Similar to the *GestureRecognizer* agent, the RoboSensor agent first has to be given an IP address of SPAK by calling the *setSPAKIP()* function.

### KnowledgeManager (SPAK)

KnowledgeManager is the intelligence part of the system. It is basically SPAK with its XML-RPC network gateway. Input data can be sent to the knowledge manager as a text message through its `setMessage()` function:

- `string setMessage(string message)`

If the message is “induce”, SPAK will start its inference process. The message can also be a knowledge query (in JavaScript), for which the result will be returned after the call has completed.

A small shell script is provided on Robovie (login: menu) to facilitate starting, stopping, and testing agents. Its screenshot is shown in Figure 6.3.

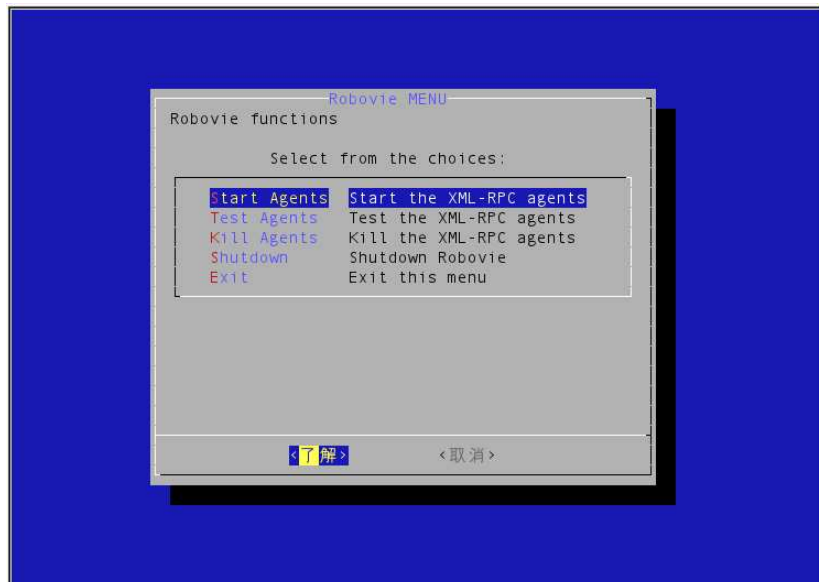


Figure 6.3: Menu selection for starting, testing and killing agents running on the Robovie robot

## 6.2 Interaction Scenarios

The demonstration prototype was set up to work in three example human-robot interaction scenarios. Figure 6.4 shows a human interacting with the robot in the demonstration scenarios. SPAK is loaded with the knowledge contents according to the design of the dialogue manager in Chapter 5. When all agents including the SPAK knowledge manager are started, the robot is ready to interact.

The first *Greeting* scenario is a multimodal state-based dialogue scenario aimed to show basic functionalities of the system in management of multi-modal dialogues and show co-operations among system components. The second scenario is the *Basic Robot Learning*. The idea is that apart from behaving according to the pre-defined behaviors during the interaction, the robot should be able adjust its behaviors from human instructions or feedback. On-line learning is considered. Given a set of sensors and actuators on the humanoid robots, the robot observes the human teaching instruction and feedback, and tries to induce new behaviors or change existing behaviors. The last *Future Welfare Robot* scenario lets the robot interact with the human in a simulated environment of service robot and the elderly person at home. The robot greets its owner, inquires the health condition, and take actions accordingly, e.g., contact the service center. This is intended to be a glimpse towards the future target welfare robot.



Figure 6.4: Human robot interaction in the scenarios

## Greeting

In this scenario, a basic multi-modal state-based interaction is demonstrated. The robot observes the existence of a human in front of it and greets him if a human is found. If the robot does not know the person, it introduces itself and asks for human name. After the human says his name, the robot saves it and associates it with his face. When the robot meets the person again, it can remember and greet him correctly.

From the robot video camera, human faces are detected and recognized by the *FaceDetector* and *FaceRecognizer* agents respectively. If the recognition result is *unknown*, the robot asks for the human name via the speech synthesis agent. The human utters his name through the microphone. The speech recognizer agent sends the recognized name to the dialogue manager. The robot then greets the human using speech and gesture. In case of first meet or wrong recognition, it adds the new face image in the database of the *FaceRecognizer* agent, thus improving its ability to recognize faces next time. The interaction scenario is as follow:

*A robot stands in the laboratory. Alex walks toward the robot. The robot spots his face and starts the conversation.*

**Robot:** Good morning. We haven't met each other before,  
have we. My name is Robovie. What's your name?  
**Alex:** Hi, my name is Alex.  
**Robot:** Nice to meet you, Alex. (performing greeting gesture)  
How are you today? David arrived at 9 am.  
**Alex:** Ah, I see. See you.

*The robot adds Alex as a new known human and updates his status of arrival. Next day the robot can remember him. Mr. Alex presents his face to the robot, it spots his face and starts the conversation.*

**Robot:** Good afternoon, Alex. How are you?

*The robot maintains the status of humans in its knowledge base. For the person it has already greeted, it uses shorter message.*

**Robot:** Hi, Alex.

*Sometimes the robot makes mistake in recognition.*

**Robot:** Good afternoon. We haven't known each other  
before, have we. My name is Robovie. What is  
your name?  
**Alex:** My name is Alex.  
**Robot:** Oh, I'm sorry I didn't recognize you. Hi, Alex,  
how are you.

Note that apart from interacting with human, the system keeps track of human's arrival time in the knowledge base. which can be queried and shared by other components of the system. With the tight integration with the knowledge base, the dialogue manager can make easily use of the knowledge provided, e.g., human status, like in this scenario.

## Basic Learning

The second scenario is aimed to show how the system changes or creates a new behavior according to human's instruction and feedback. First the robot is set up to understand some basic commands beforehand. These commands are as follows:

- *say <text>*: The robot will then utter the given <text> (using the text-to-speech software via the RobovieMouth agent).
- *look <left|center|right|up|down>*: The robot will move its neck accordingly.
- *stop*: The robot will stop any moving or speaking activities and return to the start position.
- *raise|put down <left|right|both> hand (s)*: The robot will raise and put down its hand (s) accordingly.

Based on these pre-defined commands and the simple learning algorithm [114], the robot observes human instructions (i.e., when the human commands the robot to do something) and tries to find out the concept behind those commands, i.e., why or because of which event should it do what action. A log of the interaction is as follow:

*The human subject Alex goes to the robot and starts the interaction. He points to the right. The robot notices his pointing but does nothing. Alex wants the robot to look in the direction he is pointing therefore he issues a command.*

**Alex:** Look left.

The robot follows the command by turning its face to the left. A *LearnedAction* frame is created with the action looking to the left and the triggering event of human pointing to the right (from the top of LSE list). The confidence value is set to 1. Next, Alex points to the right again. The robot still does nothing because the confidence value is less than the threshold (assumed set to 2).

**Alex:** Look left

The robot again turns to the left. This time the confidence value of the *LearnedAction* frame is increased to 2 from two teaching samples. This equals to the threshold, it becomes a new robot behavior. Now if Alex points to the right again, the robot will automatically look to the left.

## Future Welfare Robot

The last scenario simulates the interaction between a welfare robot and the human owner at home. It includes dialogue interaction to greet the human and check for his health condition. If the human exhibits some problematic symptoms then it should contact the welfare center to obtain more information. If the welfare center recommends the human to visit the center, the robot takes care of the appointment process and transportation reservation. This contains state-based and form-based types of dialogues and is intended to illustrate the future goal of welfare robots helping elderly and disabled persons living alone at home. The interaction scenario is as follow:

A welfare robot named *Robota* lives in the house with its master Alex. It is 8 o'clock in the morning. The robot detects a human face. It starts the face recognition and finds it is Alex. The robot starts the interaction.

**Robot:** Good morning, Alex [also waves its hand]. How are you today?

**Human:** Hi, Robot. Not very well.

Robot records the status of human. This value of *not very well* triggers more questions.

**R:** That is not good. Did you sleep well last night?

**H:** No.

**R:** Sleep is very important for your health. How do you feel now?

**H:** I have a headache.

With the status of not very well, and having a headache, the robot suggests to contact the welfare center.

**R:** I see, I should report to the welfare service center to get some advice for you.

[simulated] The robot contacts the welfare service center.

**R:** A physical consultant advice you to visit the center. Should I reserve a visit for you?

**H:** Yes.

To reserve a visit at the center, it needs to know the time.

**R:** Do you want to visit just now or in the afternoon?

**H:** Afternoon is better.

The robot queries the welfare center, and found the afternoon time is OK [simulated]. In order to visit the center, a transportation means is required. Therefore it reserves the bus ticket at the travel agent [simulated]. After the reservation is completed, the robot reports the task.

**R:** I have reserved a visit at 3:30 and the bus at 3:00. I will tell you when the bus arrives.

**H:** Thank you very much.

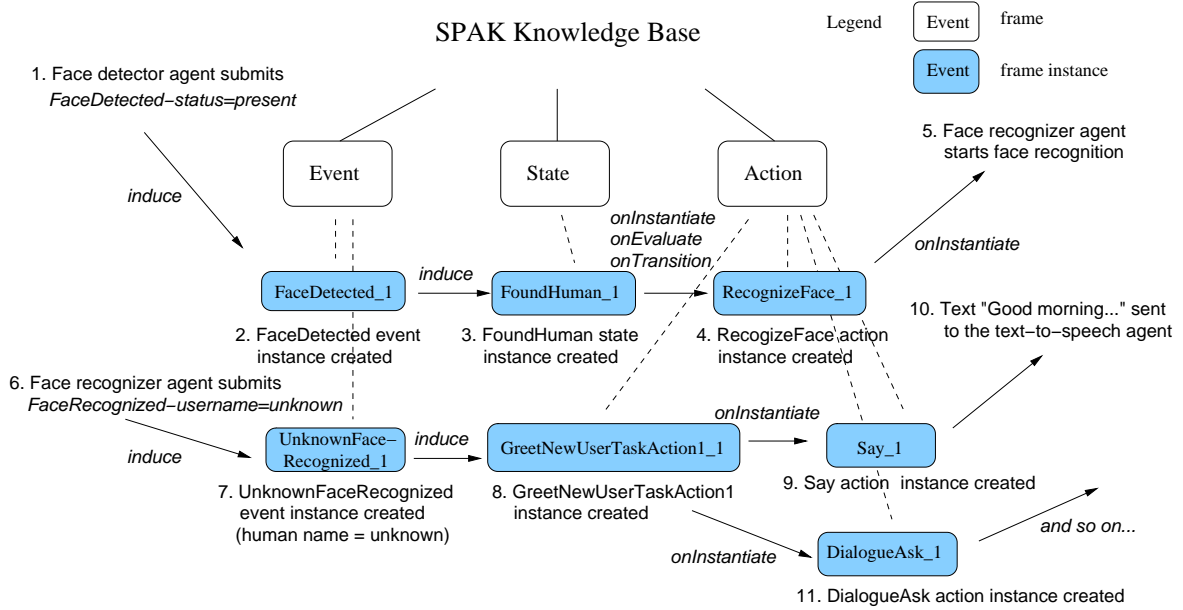
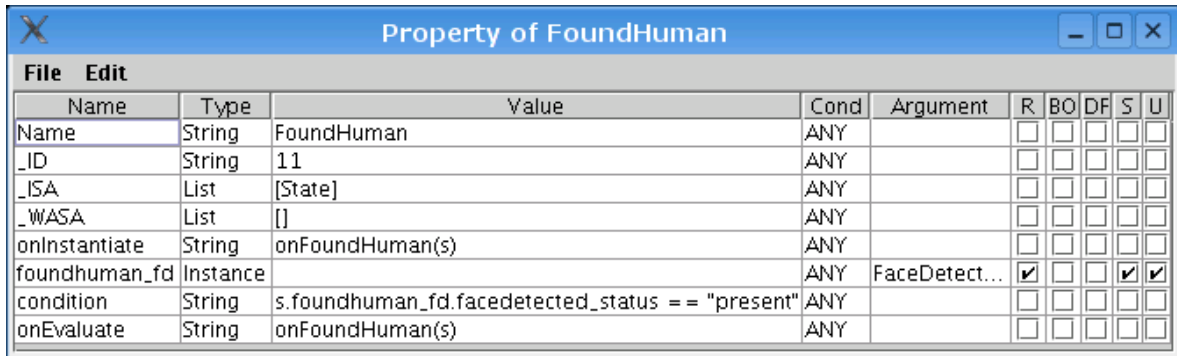


Figure 6.5: Steps showing changes in SPAK knowledge base during the human-robot interaction scenario

### 6.3 Dialogue Manager Internal Mechanisms

The *Greeting* and *Future Welfare Robot* scenarios contain state-based and frame-based dialogues. The system works according to the mechanisms described in Section 5.3. The second *Basic Learning* scenario works according to what described in Section 5.3.5. In this section, we show in details how the system works during the beginning part of the *Greeting* scenario. Changes in the system is illustrated in steps in Figure 6.5. First a human face is detected by the face detector agent (1). It submits a "*FaceDetected-status=present*" string to SPAK. Upon receiving this, an event frame *FaceDetected* is instantiated (2) and it triggers further instantiation of a state frame *FoundHuman* (3), whose property is shown in Figure 6.6. The *FoundHuman* frame has only one required slot (see the slot flag *R*), the slot *fd* of type *Instance*. This slot has a condition stating that (see the *Cond* and *Argument* column) the value must be an instance of a *FaceDetected* frame. The frame's *condition* slot is "*s.fd.status == present*" meaning that the *status* slot of the *fd* instance must be "*present*", which is just fulfilled. Therefore a *FoundHuman* state frame is instantiated. From the code specified in its *onInstantiate* slot, the function *onFoundHuman()* is executed, which will create a *RecognizeFace* action instance (4) requesting the face recognizer agent to do face recognition (5). Since it is specified similarly in the *onEvaluate* and *onTransition* slots as well, this face recognition action is also done periodically by the *Evaluator* and when the instance changes its parent.

Since *Alex* is new to the robot, the face recognition result is unknown. The text "*FaceRecognized-username=unknown*" is sent to SPAK by the face recognizer agent (6). Because of this, an event

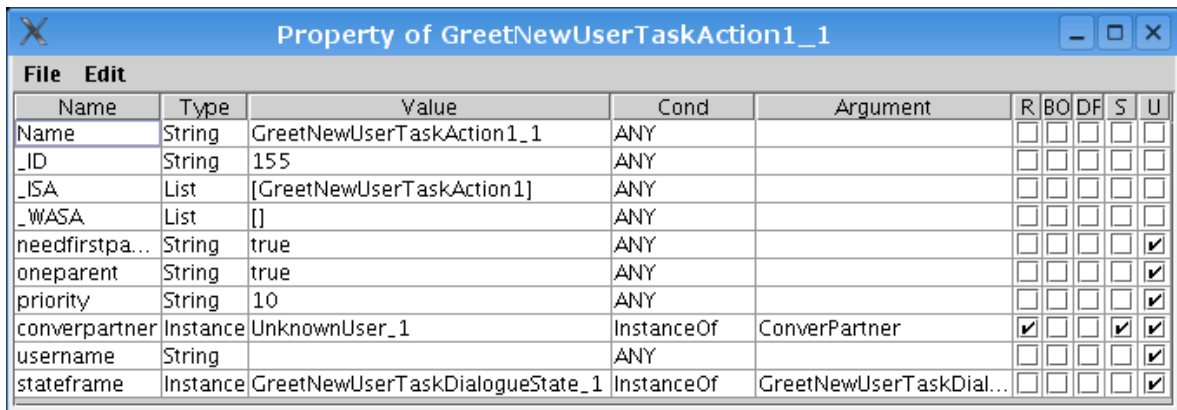


Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	FoundHuman	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	11	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[State]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
onInstantiate	String	onFoundHuman(s)	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
foundhuman_fd	Instance		ANY	FaceDetect...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
condition	String	s.foundhuman_fd.facedetected_status == "present"	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
onEvaluate	String	onFoundHuman(s)	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 6.6: A SPAK window showing property of the *FoundHuman* state frame.

from *UnknownFaceRecognized* is instantiated (7), which triggers further instantiation of an action frame instance *GreetNewUserTaskAction1* (8), starting a dialogue conversation to ask for the human's name. The properties of the *GreetNewUserTaskAction1* is shown in Figure 6.7.

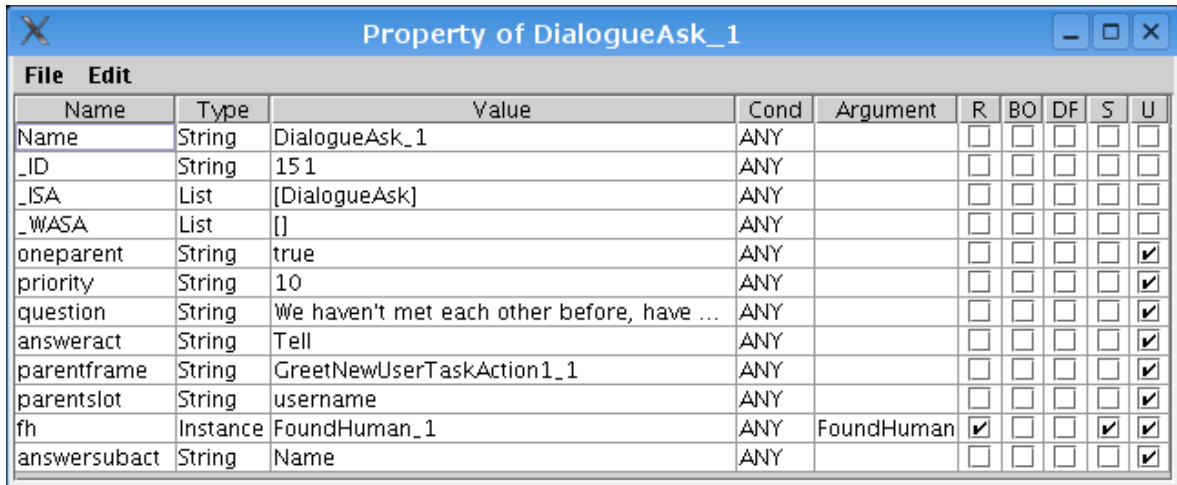
The code in *GreetNewUserTaskAction1*'s *onInstantiate* slot creates a *Say* action instance (9) to send the first greeting message "Good morning" to the human (10). In order to find out the name of this human, it also creates a *DialogueAsk\_1* frame instance (11) which will make a dialogue with the human asking for his or her name ("We haven't known....., what's your name?") and creates a *WaitingForAnswer\_1* state frame instance with expected speech act of type *Name*. A screenshot of this *DialogueAsk\_1* instance is shown in Figure 6.8.



Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	GreetNewUserTaskAction1_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	155	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[GreetNewUserTaskAction1]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
needfirstpa...	String	true	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
oneparent	String	true	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
priority	String	10	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
converpartner	Instance	UnknownUser_1	InstanceOf	ConverPartner	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
username	String		ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
stateframe	Instance	GreetNewUserTaskDialogueState_1	InstanceOf	GreetNewUserTaskDial...	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 6.7: Snapshot of a *GreetNewUserTaskAction1\_1* instance

When the human replies, the text "SpeechRecognized-text=My name is Alex" is sent from the speech recognizer agent to SPAK. A *SpeechRecognized* frame is then instantiated and the recognized text is parsed. In this case, the result speech act is *name*, which is expected by the existing *WaitingForAnswer\_1* instance. Upon this matching, an *AnswerFound* action frame is instantiated. It updates the *GreetNewUserTaskAction1\_1* instance's *username* slot and greets



Name	Type	Value	Cond	Argument	R	BO	DF	S	U
Name	String	DialogueAsk_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ID	String	151	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_ISA	List	[DialogueAsk]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
_WASA	List	[]	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
oneparent	String	true	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
priority	String	10	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
question	String	We haven't met each other before, have ...	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
answeract	String	Tell	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
parentframe	String	GreetNewUserTaskAction1_1	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
parentslot	String	username	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
fh	Instance	FoundHuman_1	ANY	FoundHuman	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
answersubact	String	Name	ANY		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 6.8: A SPAK window showing property of the *DialogueAsk\_1* instance during the interaction. It asks human according to the text in its *question* slot (*Hello, we haven't...*) and creates a *WaitingForAnswer* instance with the expected speech act according to its *answeract* and *answersubact* slots.

the user (“*Nice to meet you, Alex...*”) together with contacting the robot posture agent to make a greeting gesture. Because of this update, the code in *GreetNewUser* instance’s *onUpdate* slot will create a *Human* frame with the new name it got and sets his status to *arrived*, and notifies the face recognizer agent about the name of the face it has previously recognized as unknown. Finally all tasks are done, and the *DialogueAsk\_1*, *WaitingForAnswer\_1*, and *GreetNewUserTaskAction\_1* instances are deleted.

If the human does not respond or the response was lost, the code in *WaitingForAnswer* instance’s *onEvaluate* slot will ask again the question after a certain time limit. In case that the face recognition result is not unknown, the system greets the human accordingly and updates the slot *status*. In case of wrong recognition, it apologizes and asks the face recognizer agent to add the new face image in the database, thus improving its ability to recognize faces next time.

## 6.4 Discussions

To give an impression of the effort needed to develop a SPAK-based robot application, some statistics are given: the combined knowledge contents for the greeting and future welfare scenarios contain 117 knowledge frames and 1,598 lines of JavaScript code (476 lines of basic procedures common to all scenarios). The knowledge contents are designed to be similar to the human’s understanding of the world of interest, hence easy for humans to understand and inspect. SPAK was run on a Pentium-4 1.8 GHz machine. So far in the scenarios, the



system was operating in real time.

During the basic greeting scenario (using a simple version of the knowledge contents with 49 knowledge frames and about 800 lines of JavaScript code), there were in total 42 input events from agents, causing 45 instantiations and 19 updates of frames.

As it is tightly integrated with the knowledge base, the dialogue manager can easily make use of the knowledge provided. To illustrate the benefits, some examples can be shown as follows:

**-Report people status:** One can add new *HumanCommand* frames to answer questions like: *Is <human-name> here?*, *Who is still here?* If the *Human* frame has a slot *profession* with possible values like *student*, *professor*, and *technician*; the system can be easily extended to answer such questions like: *Are there any students coming today?* by finding all *Human* frames whose *profession* slot is *student* and *status* slot is *arrived*.

**-Resolving ambiguous requests:** It is assumed that there is a scene understanding agent that detects objects in the room and updates the knowledge in SPAK accordingly, and that the robot can pick up objects. A *PickUpObject* command frame can be added so that the robot understands command like *Give me <object>*, e.g., *Give me a pen*. Dialogue can be useful in case that the request is ambiguous. For example, if there are two pens, it can create a *DialogueAsk* frame to get the information which pen the human wants. Human can give an answer like *"the red one"*, and the robot will find an instance of pen whose color is red. Knowledge can be also useful. For example, the human asks for a green tea but there is no green tea found. Using the knowledge that green tea is a drink and a kind of tea, it can say *"I'm sorry there is no green tea available"* and propose *"Would you like another drink?"* or *"There is iced tea, would you like that instead?"*.

A SPAK-based robot application, e.g., the dialogue manager shown in this work, is a set of knowledge frames. We can run multiple applications on a single SPAK by combining their knowledge frames. Sharing of knowledge among applications can be done by sharing some knowledge frames. For example, the knowledge about a human can be shared by both dialogue manager and gesture recognizer applications. The recognized gesture is stored in a slot and can be used by the dialogue manager. On the other hand, the gesture recognizer can use the knowledge about the human when it tries to interpret the recognized gesture.

## 6.5 Summary

In this chapter we present the design and implementation of the prototype robot system. Based on the idea discussed the previous chapters, an interactive robot has been developed. Three interaction scenarios are used to show some features of our system.

## Chapter 7

# Evaluation

In this chapter we evaluate the proposed work in previous chapters. In this type of work, it is rather difficult to make a quantitative evaluation. This is similar to researches in voice processing systems, where there is difficulty finding satisfactory evaluation of the performance of the systems [71]. Common methods like performance and speed analysis do not apply here because at this stage the prototype merely serves as an illustration of concept, it is not yet in the developmental stage. Measurement of the transaction success rate or collecting opinions from humans interacting with the robot are also irrelevant because we focus on the internal design of the robot, not the dialogue interaction functionalities. Therefore we opt to provide analysis and discussions of this work by contrasting it with other related works. Three sections of this chapter discuss each of the three contributions separately.

### 7.1 SPAK Extended Frame-based System

SPAK is a modern knowledge platform targeted to link various robotics agents and applications using a blackboard architecture with intuitive user interfaces. Compared to other frame-based systems, SPAK not only supports the conventional frame model but also introduces action mechanisms through the use of special slots and flags, time-based layer, and other features in the proposed dynamic extensions. As a software package, SPAK is multi-platform, network-aware, and also features an easy-to-use graphics user interface for knowledge browsing and editing. Compared to the previous version of SPAK (discussed in [10]), updates in the present work are the new extensions: time-based layer, evaluator, and priority support, and the development of a sample dialogue system with three interaction scenarios.

Compared to the ZERO++ frame-based system [56, 115, 55], SPAK covers all ZERO++ frame formalisms and introduces more features on dynamic behaviors. Instance and vector slot types and the slot flag *R* in SPAK replace HASPARTS's FLIST-typed slot. ZERO++'s RELATIONS predicate slot is replaced by the SPAK's *condition* slot.

*FramerD* is a distributed object-oriented database designed to support the maintenance and sharing of knowledge bases [50]. It is robust, scalable, and provides good coverage of basic frame and slot operations. Actions can be associated with slots by the use of demons. However, features like GUI knowledge editor, time-based layer, and integration with robot components and agents are not provided.

*Protege* ontology development tool [116] combined with the *Algernon* rule-based inference system [117] results in a frame-based knowledge system with a GUI editor and inference engine similar to SPAK. However, as it is designed primarily for representing knowledge, it lacks support for action generation, time-based layer, and other dynamic features.

Although the development purposes of SPAK and CODE4 [51] greatly differ (SPAK is aimed to be a knowledge-based system that runs autonomously while CODE4 is targeted as a unified system for managing conceptual knowledge for average users), the design details are interestingly comparable. Many notations in CODE4 can be compared to those in SPAK. Slot names in SPAK are considered global in the reinduce process, i.e., slots of different frames that have the same (slot) name are considered the same. Although we do not define slots as frames, we can do so if needed (e.g., by using only instance type slot), and this will resemble the concepts of “properties are concepts” in CODE4. *Statement* concepts in CODE4 can be represent as Relationship frames in SPAK, with slots specifying subjects, predicates, and so on. CODE4 features a GUI user interface similar to SPAK, however, the dynamic concept proposed in this work are not considered.

Compared to the JULIA software toolkit for building embedded and distributed knowledge-based systems [52, 53], although the design and technologies used in the implementation are very similar to our system, e.g., the frame model, distributed network, Java, and XML, the main concepts are different. The purpose of the JULIA system is to solve problems of distributed knowledge sharing and reuse in intelligent information and expert systems, e.g., diagnosing and planning treatment tactics for patients in the hospital, while SPAK is aimed to be the knowledge platform for robots. The dynamic concept as proposed in this work including time-based layer, forward chaining inference, and GUI editor for easy application development are not considered in JULIA.

Table 7.1 shows a comparison of SPAK to other knowledge-based systems (the symbol ? means the information is not enough to conclude). The left-most column contains features we found useful in realizing robot systems and SPAK is developed to support all those features. The features only found in SPAK and not in other systems are time-based layer and built-in periodical task executor. Although time-related issues have been studied in the AI research community (e.g., in [118] and in modal and temporal logics), we have yet to find such features in frame-based systems.

Features	FramerD	Protege + Algernon	CODE4	JULIA	ZERO++	SPAK
Platform	Multi-platform (ANSI C)	Multi-platform (Java-based)	(McIntosh?)	Multi-platform (Java-based)	Sun, other platform (written in Common-LISP)	Multi-platform (Java-based)
Multiple Inheritance	?	No	Yes	No	No	Yes
Inference	Yes	Forward and Backward	?	Backward only	?	Forward and Backward
Procedural Script Supported	Yes	Algernon can call Java methods and internal LISP subsystem	ClearTalk script	Yes	LISP	Javascript, with access to Java API
Slot-action Mechanisms	Demons	-	?	?	-	Special slots: system slots and on-event slots
Time-based Layer	-	-	-	-	-	Yes
Built-in periodical task execution	-	-	-	-	-	Via Evaluator
GUI Editor	-	Yes	Yes	-	Yes	Yes
Network	Yes	Yes	- (web interface planned)	Yes	-	Yes
Programming Interfaces	C API, FDScript, Web	Algernon's query language	SmallTalk, CKB	Dialogue shell, Java API, CORBA	LISP	Network gateway (direct TCP, XML-RPC), Java, JavaScript API

Table 7.1: Comparison of SPAK to other frame-based systems

Some frame systems provide a mechanism to attach a procedural script to a slot (also known as active values). This script will be executed when the slot value is accessed or modified. On-event slots in SPAK, however, provide a means to specify actions to be executed in not only slot-related events (i.e., *onUpdate* slot in case of slot value updating) but also other frame-related events like frame instantiation, instance deactivation, instance evaluation, etc.

The main disadvantage of SPAK compared to those systems is that it is still in an early stage. More tests and performance improvement are needed. There are not many slot conditions, only basic operators are provided, e.g.,  $>$ ,  $<$ ,  $==$ ,  $!=$ , in (a set of possible choices), and instance-of. Also, distributed knowledge hierarchies like in JULIA are not yet supported.

SPAK can also be considered as a robot development tool. Consider recent robot development tools like the Sony Open-R [119] and Orocos [120], most of them are designed for robot control tasks, e.g., robot motion control and obstacles avoidance. These tools are suitable for behavior-based applications. However, managing human-robot interactions using natural interfaces like speech requires the machine to process symbolic information such as words and sentences used by human as well. Therefore the SPAK knowledge-based platform is definitely helpful for developing interactive robots.

## 7.2 Knowledge-based Dialogue Manager

Uses of frame-based knowledge technique in some parts of dialogue systems are not new. The common use is to store the world or domain knowledge, e.g., usage of common semantic hierarchy in TRIPS [87]. However, this does not include the discourse management and domain-specific knowledge. In the form-based approach of dialogue management (according to the categorization in [74]), frames with slots are used as dialogue forms. The general dialogue policy is to try to fill these slots. Once all slots are filled, an action like database query can be started. An example of form-based dialogue system is [78] and [80]. In Vox's FASiL system, separation of data structure that stores dialogue states from dialogue policies is proposed [70]. Frames are used to store dialogue states, and state transition network is used to specify dialogue policy. The Jaspis architecture is designed to support distributed spoken dialogs using multi-agent techniques [99]. Shared system information is organized hierarchically and made accessible via an access protocol defined using XML-DTDs. Other use of frames is, for example, in the Galaxy-II dialogue architecture, semantic frame representation is used for inter-server communications [97]. RavenClaw dialogue manager in the CMU Communicator project has another use of tree-like structure to store dialogue tasks to be executed [98]. Similar to the frame model, object oriented technique is used in the Queen's communicator [95]. Things like dialogue frames, domain experts are modeled as objects. Set of user- and database-related rules are used to manage system behaviors.

However, the uniqueness of our approach is that the single blackboard-like frame-based knowledge platform is used as the base layer in the system. In other words, the dialogue manager runs totally on the knowledge platform, not only just uses it to store some knowledge. The dialogue policy is defined using general mechanisms provided by the platform. However, it is also possible to manage dialogue interactions using external manipulator connected to SPAK. Nevertheless, the knowledge platform remains the central hub connect-

ing robotic devices, applications (e.g., dialogue management), and external manipulators (if any) together.

The main benefit of our system is that, by having the knowledge platform base, the dialogue manager integrates seamlessly with robotics devices and other robot applications. The knowledge can be easily shared among applications, avoiding integration problem in the future. Most others systems are concerned with information-giving or domain-specific planning tasks, and not well integrated with robotics devices or other applications as well as the knowledge base. Compared with them, ours is more *generic*, because the dialogue policy is based on the general and flexible mechanisms provided by the platform, and more *natural*, because in our design, interactions occur along with the changing knowledge base, which reflect meaningfully the world of interest. Also another advantage is that the total system is less complicated since a single platform serves for various robot applications. It is easy for developers to understand and improve it further. Comparison of our knowledge-based dialogue management approach with the conventional one is summarized in Table 7.2.

We showed in this thesis the dialogue manager that can handle state-based and form-based dialogue types as they are the most popular ones. However, the knowledge-based dialogue manager can be designed to support other approaches as well. For example, to support the plan-based approach, a plan engine can be attached to SPAK as an external manipulator. It receives updated knowledge from SPAK, plans new actions, updates the knowledge in SPAK accordingly, and generates output actions via SPAK (e.g., by creating Action frames).

Information state and dialogue move concepts can also be realized using SPAK. Each information state can be stored as a frame. The data structure of the Activity Model shown in [101] is very similar to the frame structure. Many components in the CIA as present in [91], e.g., dialogue move tree, active node list, pending list, and salience list, can be stored in frame hierarchies with little modification. One can add update rules that govern updating of information states using the special slots and flags provided in SPAK.

### 7.3 Distributed Knowledge-based Robot Architecture

In this section, we compare the proposed distributed knowledge-based robot architecture to some other related works. First of all, by employing a centralized symbol-based knowledge platform as the brain of the system, our architecture follows much of the blackboard design concept. The platform provides a central module which acts as blackboard, knowledge processing brain, memory, and do the judgement, task planning and execution. The platform also provides network interfaces necessary for integration of various existing modules over the network. Without such a platform, these modules will need to communicate with each other by themselves. The system will become more complex and difficult to manage as the

Conventional Approach	Knowledge-based using SPAK
<ul style="list-style-type: none"> <li>• Dialogue flows are designed according to the desired output dialogue behaviors</li> <li>• The system has no general knowledge about the world, just only the knowledge needed to accomplish certain tasks</li> <li>• Extension is needed to support knowledge sharing with other applications</li> <li>• Several databases and different data-structure for each knowledge</li> </ul>	<ul style="list-style-type: none"> <li>• Dialogue is the result of the designed actions in response to changes in the world of interest, which is represent in the knowledge base</li> <li>• The system has a detailed knowledge about the world of interest</li> <li>• The knowledge can be easily shared with other robotic applications</li> <li>• Single data base and single data-structure for knowledge representation make it easy to maintain and develop applications</li> </ul>
<ul style="list-style-type: none"> <li>• The system is optimized for the dialogue application</li> <li>• No need to maintain irrelevant knowledge</li> <li>• Many techniques, systems, and development toolkit are available</li> </ul>	<ul style="list-style-type: none"> <li>• General system for all applications, not optimized for dialogue</li> <li>• Need to maintain other knowledge that might be unnecessary for the dialogue application</li> <li>• Still in early phase, work is needed to import techniques in the conventional approach to this approach</li> </ul>

Table 7.2: Comparison of the knowledge-based dialogue management with the conventional dialogue management approach

number of modules grows.

Compared to the conventional design where system components are connected to each other in a mesh manner without a central module, the blackboard-based system allows easy control and integration of new modules into the system. Interfaces between system components are generalized. The system can perform intelligent tasks with the help of the central brain. Figure 7.1 shows a comparison of the mesh and the blackboard-based approaches. Blackboard-based design has been used in various systems and research fields, e.g., HearSay-II speech understanding system [121] and IP3S system (Integrating Process Planning and Production Scheduling) [122]. The question to such systems is what are the details inside the blackboard and how the system works to achieve the goal.

Although this thesis discusses mainly about SPAK, which is a symbol-based processing engine, our architecture is hybrid, not a pure deliberate one. The multi-layered intelligent

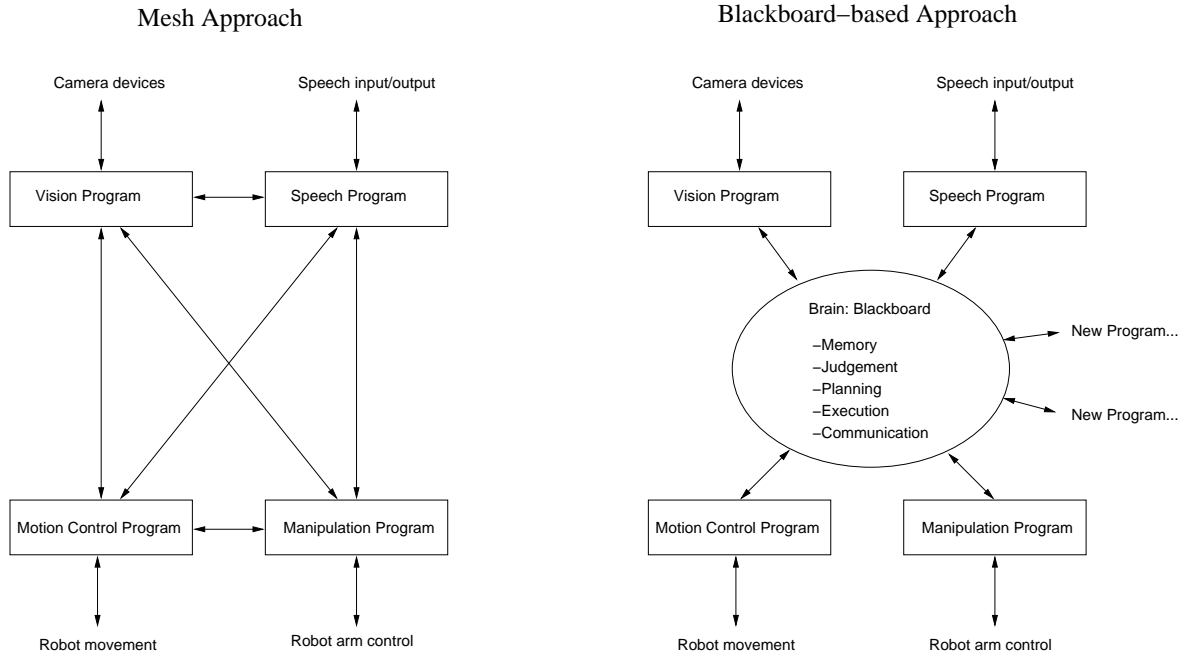


Figure 7.1: Blackboard-based approach (right) to develop robots where a central brain is employed, compared to the conventional mesh design (left)

control system, though not a focus in this thesis, is implemented by a group of networked software agents. Each agent represents a processing element or the robot and is in charge of lower-layer control such as obstacle avoidance which requires immediate response. Higher level controls are achieved by communication between the agents and the central frame-based knowledge module SPAK.

Compared to the architecture proposed in [34, 25], which is based on state-transition network of situated-modules, although the robot apparently interacts with humans according to the network, it lacks the knowledge about the world it is in, which is needed in order to achieve more intelligent behaviors. There is no use of a knowledge base in the system. In our architecture, we use hierarchies of frames to represent the world of interest and manage robot behaviors. Frame hierarchies represent meaningfully the changing world and necessary knowledge for managing robot behaviors. Subset of which are event and action frames, which can be designed so that they generate robot behaviors, e.g., following human commands and making dialogue interaction. Why a certain robot action is made can be rationally and easily explained by those knowledge frames. The up-to-date representation of the world in the knowledge base can be useful in development of other intelligence behaviors.

Compared to the Intelligent Machine Architecture (IMA) by Kawamura et al. [38], although we share some similarities by using agents to represent robot components, usage of knowledge base is not concerned in IMA. There is a use of knowledge technique in the Carl



robot [39] in ontologies which organize and compose behaviors as a part of robot's innate knowledge [40], but not as the main processing engine as in ours. The dialogue manager of the Jijo-2 robot supports frame-based (form-based) dialog and the current dialogue state and a salient entities list that contains entities referred to by the preceding utterances are maintained in the system, similar to the use of status registers including LSE and LSA in our system. However, usage of a knowledge base is not concerned. Lacking of a firm knowledge base infrastructure means an extra effort is needed to achieve further intelligence behaviors.

Compared to other robot systems that have hybrid architectures, e.g., Situated-module-based architecture [25], Tagged-behavior-based architecture [27], and the architecture by Lopes et al. in [40], although these systems are effective in individual applications they are targeted (e.g., dialogue management, behaviors switching, and path learning from human instruction), it is difficult to add new applications into the system and share knowledge among applications. The underlying architecture is usually influenced by target applications and the system components. In case that the target goal is changed, which might result in adding new applications or new components to the system, the architecture might need to be adjusted. When an architecture is very application-oriented, developers have to learn specific features of that certain system. Hence, designing an application is more complicated and not intuitive, compared to systems with application-independent general design.

The fact that frame-based knowledge platform is used makes our architecture fundamentally different from other architectures, e.g., Care-O-bot, which employs plan-based hybrid architecture [123]; Godot robot [124], which uses a logic-based TRINDI dialogue move engine [89]; and the Instruction-Based Learning (IBL) robot, which uses Python language script to store learned task knowledge [16]. However, our frame-based approach does not reject such techniques, as they have advantages in different kinds of tasks, e.g., planning tasks using plan-based and problem solving using logic-based techniques. We envision that they can be integrated<sup>1</sup> with the frame model as the base layer linking other techniques, because of the naturalness and simplicity of the frame model. The knowledge base is placed in the center of the system and other techniques to be included in the system must interact with it using provided means and update the knowledge part they are responsible for. This means that the knowledge contents are always up-to-date for all techniques and possible inter-operation problems with using different techniques are eliminated.

---

<sup>1</sup>One can also implement some other techniques, e.g., rule-based, plan-based, directly in SPAK using the frame-based infrastructure provided, though. However, in this case we assume it is not desired or rather difficult to do so, and one wants to simply use other tools and SPAK in the system.

## Chapter 8

# Conclusions and Future Work

We introduced in this thesis a multi-agent knowledge-based robot architecture featuring the frame-based knowledge software platform SPAK. The originality of this work is the SPAK's extended frame-based knowledge model and its application of a knowledge-based dialogue manager for interactive robots. New extensions namely time-based layer, evaluator, and priority support are introduced to the frame model to support representation of dynamic data and management of robot behaviors.

With SPAK it is easier to manage multi-modal human-robot interactions using a black-board mechanism. Various robotic applications can run cooperatively in SPAK and share knowledge among each other. The frame-based knowledge platform makes no assumption concerning the applications running on it. Hence, instead of having a dedicated system for each robot application, the single knowledge platform serves as a basis infrastructure for all, ensuring interoperability among them and easy sharing of knowledge. Knowledge sharing among applications can be done by sharing knowledge frames. Based on the concept that the world of interest is meaningfully represented in the frame hierarchies, developers can easily and intuitively design robot applications using the general and flexible mechanisms provided by the platform. By introducing the multi-agent network architecture, it is easy to collaborate with remote agents, e.g., service agents at the welfare center.

SPAK knowledge editor allows simple and intuitive development of robot applications. Our SPAK-based robot system has an important feature that frames and agents are integrated seamlessly for dynamic robotic control in a distributed environment. A prototype system with a sample dialogue management application has demonstrated interesting functions for future symbiotic robots.

### 8.1 Contributions

This chapter reviews the dissertation's contribution to the fields of robotics, knowledge engineering, and dialogue systems:

- **Extended Frame-based Knowledge Model:** we proposed dynamic extensions to the conventional frame model and realized it in our in-house SPAK software platform.
- **Knowledge-based Dialogue Manager:** we proposed a novel design of a dialogue manager that is based completely on the knowledge platform, not just using the knowledge base to store some data.
- **Multi-Agent Knowledge-based Robot Architecture:** based on the previous two points, we proposed an architecture for interactive robots and demonstrated a working system on a humanoid robot.

## 8.2 Limitations and Future Directions

Achieving a fully-functional symbiotic robot is a multi-disciplinary and long-term task. Works in many parts need to be done in parallel and combined to achieve a functional integrated system. For this kind of task, the glue is very important. There are so many parts of the system that need to be integrated together. These include robot hardware and software parts, which have different interfaces and usages, depending on the makers, versions, etc., and variety of robot applications, depending on what kinds of behaviors we expect the robot to do.

In this work we proposed the glue that is a multi-agent robot architecture based on a knowledge-based platform and showed a reference implementation in SPAK and the Robovie-II humanoid robot. Future work on the platform includes SPAK performance analysis and optimization. For the time-based layer support, history data — although now the data is saved only when a change occurs —, will accumulate and consume an increasing amount of memory. A mechanism to purge unused data to a permanent storage and retrieve them back efficiently when needed is required. Another possible enhancement of SPAK is to include support for other AI techniques like plan-based and logic-based techniques. For example, knowledge frame hierarchies in SPAK can be exported to or let manipulated by the JESS Java-based rule engine [125] for tasks like planning. Moreover, although currently the knowledge contents are stored only in a single knowledge base in order to avoid the knowledge-coherence problem, the distributed knowledge model (e.g., in [53]) might also be considered.

As the current dialogue manager is designed just to illustrate features of the system, we made assumptions that there is no speech recognition and understand errors, and human understands the limitation of the system in making dialogue conversations. For the more realistic usage, techniques from Natural Language Processing (NLP) and spoken dialogue systems fields can be introduced to cope with such errors and other ambiguities as well as deviation from the defined paths by the human conversation partner. Also other aspects

of dialogue communications like error recovery, topic switching, synchronization among multimodal inputs need to be taken care of. Supports of other dialogue models apart from the state-based and form-based dialogue types, e.g., the information-state model, can be added. One possibility is to port elements of dialogue manager in the CIA architecture [91] to the frame model.

The current system can be enhanced with more robot devices and applications. For the robot devices developers, they need to make sure that the new or upgraded devices will provide access interfaces which are accessible by other agents on the system. For robot applications designers and developers, they have to make sure that the new application to be on the robot (i.e., in SPAK) makes use of and updates the knowledge base properly, as the knowledge base is the main communications means among application. The knowledge base itself also needs to be improved in case that limitations are discovered when adding new robotic devices and applications, or setting up the system in new interaction scenarios.

Although development in each part can proceed independently, all parts need to be integrated and tested from time to time to make sure there are no wrong assumptions made in any parts, since the design relies heavily on symbols and abstraction.

SPAK has been used in robot applications like scene understanding and task planning [10] and is used in the ongoing work of dialogue management, gesture-based human robot interaction [108], and multi-robot collaboration [109]. As the target symbiotic robot might need to navigate around, and manipulate things to serve its master, more applications from the robotics side, e.g., robot navigation and manipulation control, are desired.

Latest information on the SPAK tool is available on its home page [110].

### 8.3 Concluding Remarks

This work presents the design and development of interactive robots based on knowledge and multi-agent techniques. The design is modular, new robotic devices can be added easily. By using the knowledge platform, the robot has an up-to-date, easy-to-understand unified representation of the world. Robot applications can easily make use of this information and share it with other applications.

## About Author

<b>Name</b>	Pattara Kiatisevi
<b>Birthdate</b>	April 14, 1976
<b>Birthplace</b>	Bangkok, Thailand
<b>Nationality</b>	Thai
<b>Status</b>	Single
<b>Educations</b>	2000/05-2002/06 – M. Sc. in Information Technology, University of Stuttgart, Germany
	1992/05-1996/04 – B. Eng. in Electrical Engineering, Chulalongkorn University, Bangkok
	1990/05-1992/04 – Triamudomsuksa high school, Bangkok, Thailand
	1987/05-1990/04 – Satit Pratumwan demonstration school, Bangkok, Thailand
	1981/05-1987/04 – Phyathai primary school, Bangkok, Thailand
<b>Experiences</b>	2000/09-2002/07 – Research Assistant, Computer Architecture Department, Institute of Computer Science, University of Stuttgart, Germany 1996/04-2000/01 – Assistant Researcher, Network Technology Laboratory, National Electronics and Computer Technology Center (NECTEC), Bangkok, Thailand

## Related Publications

1. **P. Kiatisevi**, V. Ampornaramveth, H. Ueno, "*A Frame-based Knowledge Software Tool for Developing Interactive Robots*", To appear in the Journal of Artificial Life and Robotics Vol. 9, Springer
2. **P. Kiatisevi**, V. Ampornaramveth, H. Ueno, "*Dialog Manager for Robots using Frame-based Knowledge Platform*", Proc. of the 2nd International Workshop on Man-Machine Symbiotic Systems, Kyoto, November, 2004
3. **P. Kiatisevi**, V. Ampornaramveth, H. Ueno, "*Knowledge-based Interactive Robot: System Architecture and Dialogue Manager*", Proc. of the 8th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2004), Auckland, New Zealand, August 2004
4. **P. Kiatisevi**, V. Ampornaramveth, H. Ueno, "*A Distributed Architecture for Knowledge-based Interactive Robots*", Proc. of the 2nd International Conference on Information Technology for Application (ICITA 2004), Harbin, China, January, 2004
5. V. Ampornaramveth, **P. Kiatisevi**, H. Ueno, "*SPAK: Software Platform for Agents and Knowledge Management in Symbiotic Robots*", IEICE Trans. Information and Systems, Vol.E87-D No.4, pp-886-895, 2004
6. T. Zhang, V. Ampornaramveth, **P. Kiatisevi**, Md. Hasanuzzaman, H. Ueno, "*Coordinative Control of Multi-robot System by means of Software Platform of Agents and Knowledge Management*", Technical report of IEICE 2003-51, 2004
7. Md. Hasanuzzaman, T. Zhang, V. Ampornaramveth, **P. Kiatisevi**, Y. Shirai, H. Ueno, "*Gesture based human-robot interaction using a frame based software platform*", Proc. of the Intl. Conf. on Systems Man and Cybernetics (IEEE SMC 2004), The Netherlands, 2004.
8. V. Ampornaramveth, **P. Kiatisevi**, H. Ueno, "*Toward a Software Platform for Knowledge Management in Human-Robot Environment*", Technical Report of IEICE, Vol. 103 No. 83, pg. 15-20, 2003.
9. V. Ampornaramveth, **P. Kiatisevi**, S. Kuromiya, Y. Isoda, S. Kurakake, H. Ueno, "*Intelligent Gourmet Advisor System for Mobile Users*", Technical Report of IEICE, Vol. 102 No.

702, 2003

10. V. Ampornaramveth, **P. Kiatisevi**, H. Ueno, "*Knowledge Management Platform for Symbiotic Robots*", RSJ 2003, The Robotics Society of Japan, 2003.

# Bibliography

- [1] Ministry of Public Management, Home Affairs, Posts, and Telecommunications. Japan Statistical Yearbook 2003. [Online]. Available: <http://www.stat.go.jp/>
- [2] H. Ueno, "Symbiotic Information Systems: Towards an Ideal Relationship of Human-Beings and Information Systems," *Technical Report of IEICE, KBSE2001-15:27-34*, August 2001.
- [3] —, "A cognitive science-based knowledge modeling for autonomous humanoid service robot towards a human-robot symbiosis," *Frontiers in Artificial Intelligence and Applications*, vol. 67, pp. 123–136, 2001.
- [4] V. Ampornaramveth and H. Ueno, "Concepts of symbiotic information system and its application to robotics," *Information Modelling and Knowledge Bases XIII, H. Kangassalo et al. (Eds.)*, 2002.
- [5] The Engineering Academy of Japan, "Living with Robots - Symbiosis of Robots and Human Beings, Tokyo, Japan," 2004.
- [6] K. Kawamura and T. Davis, "International Workshop on Biorobotics: Human-Robot Symbiosis, Tsukuba, 1995," *Robotics and Autonomous Systems*, vol. 18, no. 1-2, pp. 1–291, 1996.
- [7] T. Matsuyama, "Preface," in *Proc. of the 1st International Workshop on Man-Machine Symbiotic Systems, Kyoto*, 2002.
- [8] H. Ueno, "Symbiotic robotics project." [Online]. Available: <http://research.nii.ac.jp/ueno/>
- [9] T. Fong, I. Nourbakhsh, K. Dautenhahn, "A survey of socially interactive robots," *Robotics and Autonomous Systems*, vol. 42, 2002. [Online]. Available: [citeseer.nj.nec.com/fong03survey.html](http://citeseer.nj.nec.com/fong03survey.html)
- [10] V. Ampornaramveth, P. Kiatisevi, and H. Ueno, "SPAK: Software Platform for Agents and Knowledge Management in Symbiotic Robots," *IEICE Trans. Information and Systems, Vol.E87-D No.4*, pp. 886–895, 2004.



- [11] M. Minsky, "A framework for representing knowledge," *MIT-AI Laboratory Memo 306*, 1974.
- [12] N. F. Noy, M. A. Musen, J. L. V. Mejino, Jr., C. Rosse, "Pushing the envelope: challenges in a frame-based representation of human anatomy," *Data Knowledge Engineering*, vol. 48, no. 3, pp. 335–359, 2004.
- [13] H. Ueno and Y. Saito, "Model-based vision and intelligent task scheduling for autonomous human-type robot arm," *Robotics and Autonomous System*, vol. Special Issue of Robotics and Autonomous System, pp. 195–206, 1996.
- [14] G. Bekey, *Autonomous robots : from biological inspiration to implementation and control*. MIT Press, 2005.
- [15] P. Makowski, "Survey of architectures and frameworks for autonomous agents," in *Agrobotics Workshop 2004, Horsens, Denmark*, 2004. [Online]. Available: <http://www.agrobotics.dk/workshop2004.htm>
- [16] S. Lauria, G. Bugmann, T. Kyriacou, and E. Klein, "Mobile robot programming using natural language," *Robotics and Autonomous Systems*, 38 (3/4), pp. 171–181, 2002.
- [17] A. Cangelosi and S. Harnad, "The adaptive advantage of symbolic theft over sensorimotor toil: Grounding language in perceptual categories," *Evolution of Communication*, vol. 4, no. 1, pp. 117–142, 2001. [Online]. Available: <http://www.tech.plym.ac.uk/soc/research/neural/staff/acangelosi/papers/cangelosi-evocom.ps.zip>
- [18] S. Coradeschi and A. Saffiotti, "An introduction to the anchoring problem," *Robotics and Autonomous Systems*, vol. 43, no. 2-3, pp. 85–96, 2003, special issue on perceptual anchoring. Online at <http://www.aass.oru.se/Agora/RAS02/>.
- [19] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automations*, vol. RA-2, no. 1, 1986.
- [20] —, "Intelligence without representation," *Int. J. Artificial Intelligence*, pp. Vol 48, pp. 139–159, 1991.
- [21] N. R. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 7–38, 1998. [Online]. Available: [citeseer.ist.psu.edu/jennings98roadmap.html](http://citeseer.ist.psu.edu/jennings98roadmap.html)
- [22] M. J. Mataric, "Behavior-based robotics," *MIT Encyclopedia of Cognitive Sciences (R. A. Wilson and F. C. Keil, Eds.)*, pp. 74–77, 1999.

- [23] L. Seabra Lopes, J.H. Connell, "Semisentient robots: Routes to integrated intelligence," *Semisentient Robots (special issue of IEEE Intelligent Systems, vol. 16, n. 5), Computer Society*, p. 10-14., 2001.
- [24] J. H. Connell, "Sss: A hybrid architecture applied to robot navigation," *Proc. of the 1992 IEEE Conference on Robotics and Automation (ICRA-92)*, pp. 2719-2724, 1992.
- [25] H. Ishiguro, T. Kanda, K. Kimoto, and T. Ishida, "A robot architecture based on situated modules," in *Proceedings of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*. IEEE, 1999, pp. 1617-1623.
- [26] R.T. Pack, D.M. Wilkes, K. Kawamura, "A Software Architecture for Integrated Service Robot Development," in *IEEE Conf. On Systems, Man, and Cybernetics*, 1997, pp. 3774-3779.
- [27] I. Horswill, "Tagged behavior-based architectures: Integrating cognition with embodied activity," *IEEE Intelligent Systems*, vol. 16, no. 5, pp. 30-38, 2001.
- [28] D. J. Scales and M. S. Lam, "The design and evaluation of a shared object system for distributed memory machines," in *Proc. of OSDI'94*, 1994.
- [29] avec L. Brunie and O. Reymann, "Dosmos+ : Scalable distributed shared memory environment including monitoring facilities," in *Parallel Programming Environments for High Performance Computing*. Alpes d'Huez, Avril, 1996, pp. 165-168.
- [30] Object Management Group, Inc., "CORBA OMG/ISO Standards," June 2005. [Online]. Available: <http://www.corba.org/standards.htm>
- [31] UserLand Software, Inc., "XML-RPC," June 2005. [Online]. Available: <http://www.xmlrpc.org/>
- [32] W3C, "Simple Object Access Protocol (SOAP) Specification," June 2005. [Online]. Available: <http://www.w3.org/TR/soap/>
- [33] The Foundation for Intelligent Physical Agents. FIPA Web Site. [Online]. Available: <http://www.fipa.org/>
- [34] T. Kanda, H. Ishiguro, M. Imai, T. Ono, and K. Mase, "A constructive approach for developing interactive humanoid robots," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS 2002)*, 2002, pp. 1265-1270.
- [35] T. Kanda, H. Ishiguro, T. Ono, M. Imai, and R. Nakatsu, "Development and evaluation of an interactive humanoid robot: Robovie," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2002)*. IEEE, 2002.

- [36] T. Matsui, H. Asoh, J. Fry, Y. Motomura, F. Asano, T. Kurita, I. Hara, N. Otsu, "Integrated natural spoken dialogue system of jijo-2 mobile robot for office services," in *AAAI/IAAI*, 1999, pp. 621–627. [Online]. Available: [citeseer.nj.nec.com/matsui99integrated.html](http://citeseer.nj.nec.com/matsui99integrated.html)
- [37] J. Fry, H. Asoh, and T. Matsui, "Natural dialogue with the jijo-2 office robot," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-98)*, 1998.
- [38] K. Kawamura, R. Peters, R. Bodenheimer, N. Sarkar, J. Park, C. Clifton, A. Spratley, K. Hambuchen, "A parallel distributed cognitive control system for a humanoid robot," *International Journal of Humanoid Robotics*, vol. 1, no. 1, pp. 65–93, 2004.
- [39] L. S. Lopes, "Carl: from situated activity to language level interaction and learning," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems, (IROS)*, 2002.
- [40] L. S. Lopes and A. Teixeira, "Human-robot interaction through spoken language dialogue," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS-2000)*. IEEE, 2000.
- [41] S. Thrun, M. Bennewitz, W. Burgard, A.B. Cremers, Frank Dellaert, Dieter Fox, D. Haehnel, Chuck Rosenberg, Nicholas Roy, Jamieson Schulte, D. Schulz, "Minerva: A second generation mobile tour-guide robot," in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA'99)*, 1999.
- [42] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, S. Thrun, "The interactive museum tour-guide robot," in *Proc. of the National Conference on Artificial Intelligence*, 1998.
- [43] N. Etani, "Robot media communication: An interactive real-world guide agent," in *Proc. of the 1st International Symposium on Agent Systems and Applications/3rd International Symposium on Mobile Agents (ASA/MA '99)*, 1999, pp. 234–241.
- [44] D. Koller and A. Pfeffer, "Probabilistic frame-based systems," in *Proc. of the 15th National Conference on AI (AAAI'98)*, July 1998, pp. 580–587.
- [45] P. H. Winston, *Artificial Intelligence*. Addison-Wesley, 1977, second printing, 1979.
- [46] P. D. Karp, "The design space of frame knowledge representation systems." in *Technical Report 520, SRI International Artificial Intelligence Center*, 1993. [Online]. Available: [citeseer.ist.psu.edu/karp93design.html](http://citeseer.ist.psu.edu/karp93design.html)
- [47] S. Russell and P. Norvig, *Artificial Intelligent A Modern Approach*. Prentice Hall, 2002.

- [48] V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, and J. Rice, "OKBC: A programmatic foundation for knowledge base interoperability," in *AAAI/IAAI*, 1998, pp. 600–607. [Online]. Available: [citeseer.ist.psu.edu/chaudhri98okbc.html](http://citeseer.ist.psu.edu/chaudhri98okbc.html)
- [49] N. Noy, R. Fergerson, and M. Musen, "The knowledge model of protege-2000: Combining interoperability and flexibility," 2000. [Online]. Available: [citeseer.ist.psu.edu/noy01knowledge.html](http://citeseer.ist.psu.edu/noy01knowledge.html)
- [50] Beingmeta, "FramerD," June 2005. [Online]. Available: <http://www.framerD.org/>
- [51] D. Skuce and T. C. Lethbridge, "CODE4: a unified system for managing conceptual knowledge," *Int. J. Hum.-Comput. Stud.*, vol. 42, no. 4, pp. 413–451, 1995.
- [52] D. Soshnikov, "Software toolkit for building embedded and distributed knowledge-based systems," in *Proc. of the 2nd Intl. Workshop on Computer Science and Information Technologies CIST'2000*, 2000.
- [53] —, "An architecture of distributed frame hierarchy for knowledge sharing and reuse in computer networks," in *In Proc. of the 2002 IEEE International Conference on Artificial Intelligence Systems*. IEEE, 2002, pp. 115–119.
- [54] T. Winograd, "Procedures as a representation for data in a computer program for understanding natural language," *MIT AI Technical Report 235*, 1971.
- [55] H. Ueno, "A knowledge-based information modeling for autonomous humanoid service robot," *IEICE Trans. on Information and Systems*, vol. E85-D, no. 4, pp. 657–665, 2002.
- [56] H. Ueno, S. Kogure, and Y. Ookuma, "Implementation of frame-based knowledge engineering environment zero in c++," in *Proc. of FOSE 1995*, 1995, pp. 101–110.
- [57] P. Cohen, "Dialogue modeling," in *Survey of the State of the Art in Human Language Technology*. Cambridge University Press, 1996.
- [58] E. Levin, R. Pieraccini, W. Eckert, P. D. Fabbriozio, and S. Narayanan, "Spoken language dialogue, from theory to practice," in *Proc. IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU*, 1999.
- [59] E. Levin, R. Pieraccini, and W. Eckert, "Using markov decision process for learning dialogue strategies," in *Proc. ICASSP 98, Seattle, WA*, 1998.
- [60] K. Komatani, S. Ueno, T. Kawahara, and H. G. Okuno, "User Modeling in Spoken Dialogue Systems for Flexible Guidance Generation," in *EUROSPEECH*, 2003.

- [61] V. Zue, et al., "Jupiter: A telephone-based conversational interface for weather information," *IEEE Trans. on Speech and Audio Processing*, Vol. 8 , No. 1, 2000.
- [62] B. Pellom, W. Ward, J. Hansen, K. Hacioglu, J. Zhang, X. Yu, and S. Pradhan, "University of colorado dialog systems for travel and navigation," in *Proceedings of the Human Language Technology Conference (HLT-2001)*. Association for Computational Linguistics, 2001.
- [63] W. Wahlster, N. Reithinger, and A. Blocher, "Smartkom: Multimodal communication with a life-like character," in *Proc. of Eurospeech 2001, Aalborg (Denmark)*, 2001.
- [64] M. Johnston, S. Bangalore, G. Vasireddy, A. Stent, P. Ehlen, M. Walker, S. Whittaker, and P. Maloor, "Match: An architecture for multimodal dialogue systems," in *Proc. of ACL 2002*, 2002.
- [65] A. Stent, J. Dowding, J. Gawron, E. O. Bratt, and R. Moore, "The commandtalk spoken dialogue system," in *Proc. of the ACL 1999*, 1999.
- [66] N. Dahlbck, A. Flycht-Eriksson, A. Jnsson, and P. Qvarfordt, "An architecture for multi-modal natural dialogue systems," in *Proc. of ESCA Tutorial and Research Workshop (ETRW) on Interactive Dialogue in Multi-Modal Systems, Germany*, 1999.
- [67] J. Gustafson, N. Lindberg, and M. Lundeberg, "The august spoken dialogue system," in *Proc. of Eurospeech 1999*, 1999.
- [68] J. Cassell, T. Bickmore, L. Cambell, K. Chang, H. Vilhjalmsson, and H. Yan, "Requirements for an architecture for embodied conversational characters," in *Proc. of Computer Animation and Simulation 1999*, 1999.
- [69] J. R. Searle, *Speech Acts: An essay in the philosophy of language*. Cambridge University Press, 1976.
- [70] K. Robinson, D. Horowitz, E. Bobadilla, M. Lascelles, A. Suarez, "Conversational dialogue management in the fasil project," in *In Proc. of SIGdial Workshop*, 2004.
- [71] E. Giachin, "Spoken language dialogue," in *Survey of the State of the Art in Human Language Technology*. Cambridge University Press, 1996.
- [72] G. Churcher, "Dialogue management systems: a survey and overview," 1997. [Online]. Available: [citeseer.ist.psu.edu/churcher97dialogue.html](http://citeseer.ist.psu.edu/churcher97dialogue.html)
- [73] W. Xu, B. Xu, T. Huang, and H. Xia, "Bridging the gap between dialogue management and dialogue models," in *Proceedings of the 3rd SIGdial Workshop on Discourse and Dialogue*, 2002.

- [74] M. F. McTear, "Spoken dialogue technology: enabling the conversational user interface," *ACM Computing Surveys*, vol. 34, pp. 90 - 169, 2002.
- [75] J. Allen, G. Ferguson, and A. Stent, "An architecture for more realistic conversational systems," in *Proc. of IUI 2001, Santa Fe, New Mexico, USA*, 2001.
- [76] S. Sutton, R. Cole et al., "Universal speech tools: the cslu toolkit," in *Proc. of the International Conference on Spoken Language Processing (ICSLP), Sydney, Australia*, 1998, pp. 3221-3224. [Online]. Available: [citeseer.ist.psu.edu/sutton98universal.html](http://citeseer.ist.psu.edu/sutton98universal.html)
- [77] J. Glass and E. Weinstein, "SPEECHBUILDER: Facilitating Spoken Dialogue System Development," in *Proc. European Conference on Speech Communication and Technology*, Aalborg, Denmark, September 2001, pp. 1335-1339.
- [78] D. Goddeau, H. Meng, J. Polifroni, S. Seneff, S. Busayapongchai, "A form-based dialogue manager for spoken language applications," in *In Proc. of ICSLP*, 1996.
- [79] A. Rudnicky, E. Thayer, P. Constantinides, C. Tchou, R. Shern, K. Lenzo, W. Xu, and A. Oh, "Creating natural dialogs in the carnegie mellon communicator system," in *Proc. of Eurospeech*, 1999, 1999, pp. 1531-1534.
- [80] P. Madeira, M. Mouro, Nuno Mamede, "STAR - A Multiple Domain Dialog Manager," in *5th Intl. Conf. on Enterprise Information Systems - Angers, France*, 2003.
- [81] W3C, "VoiceXML Specification Version 2.0." [Online]. Available: <http://www.w3.org/TR/voicexml20/>
- [82] VoiceXML Forum, "VoiceXML Web site." [Online]. Available: <http://www.voicexml.org/>
- [83] P. Cohen and C. R. Perrault, "Elements of a plan-based theory of speech acts," *Cognitive Science*, vol. 3, no. 3, pp. 177-212, 1979.
- [84] J. Allen and C. R. Perrault, "Analyzing intention in utterances," *Artificial Intelligence*, vol. 15, no. 3, pp. 143-178, 1980.
- [85] R. A. Cole et al., *Survey of the State of the Art in Human Language Technology*. Cambridge University Press, 1996.
- [86] J. Chu-Carroll and S. Carberry, "Collaborative response generation in planning dialogues," *Comput. Linguist.*, vol. 24, no. 3, pp. 355-400, 1998.
- [87] J. Allen, D. Byron, M. Dzikovska, G. Ferguson, L. Galescu, A. Stent, "An architecture for a generic dialogue shell," *NLENG: Natural Language Engineering, Cambridge University Press*, vol. 6, 2000. [Online]. Available: [cite-seer.ist.psu.edu/allen00architecture.html](http://citeseer.ist.psu.edu/allen00architecture.html)

- [88] D. Traum, J. Bos, R. Cooper, S. Larsson, I. Lewin, C. Metheson, and M. Poesio, "A model of dialogue moves and information state revision," *Technical Report D2.1, Trindi*, 1999.
- [89] S. Larsson and D. Traum, "Information state and dialogue management in the trindi dialogue move engine toolkit," *Natural Language Engineering*, vol. 6, pp. 323–340, 2000.
- [90] J. Bos, E. Klein, O. Lemon, and T. Oka, "Dipper: Description and formalisation of an information-state update dialogue system architecture," in *Proc. of the 4th SIGdial Workshop on Discourse and Dialogue (SIGDIAL)*, Sapporo, Japan, 2003.
- [91] O. Lemon and A. Gruenstein, "Multithreaded context for robust conversational interfaces: Context-sensitive speech recognition and interpretation of corrective fragments," *ACM Trans. Comput.-Hum. Interact.*, vol. 11, no. 3, pp. 241–267, 2004.
- [92] R. Smith and D. R. Hipp, *Soken Natural Language Dialog Systems: A Practical Approach*. Oxford University Press, New York, 1994.
- [93] O. Lemon, L. Cavedon, B. Kelly, "Managing Dialogue Interaction: A Multi-Layered Approach," in *Proceedings of the 4th SIGdial Workshop on Discourse and Dialogue*, 2003.
- [94] W. Wahlster (Editor), *Verbmobil: Foundations of Speech-to-Speech Translation*. Springer, 2001.
- [95] I. O'Neill, P. Hanna, X. Liu, M. McTear, "The queen's communicator: An object-oriented dialogue manager," in *Proc. of EUROSPEECH*, 2003.
- [96] DARPA Communicator Project, "Galaxy Architecture Web Site," June 2005. [Online]. Available: <http://communicator.sf.net>
- [97] S. Seneff, Ed. Hurley, R. Lau, C. Pao, P. Schmid, V. Zue, "Galaxy-ii: A reference architecture for conversational system development," in *In Proc. of ICSLP*, 1998.
- [98] D. Bohus, A. Rudnicky, "Ravenclaw: Dialog management using hierarchical task decomposition and an expectation agenda," in *Proc. of EUROSPEECH*, 2003.
- [99] M. Turunen, J. Hakulinen, "Jaspis - a framework for multilingual adaptive speech applications," in *Proc. of ICSLP*, 2000.
- [100] A. Cheyer and D. Martin, "The open agent architecture," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 4, no. 1, pp. 143–148, March 2001, oAA.
- [101] O. Lemon, A. Gruenstein, and S. Peters, "Collaborative activities and multi-tasking in dialogue systems," *Traitement automatique des langues, Special issue on dialogue*, vol. 43, no. 2, pp. 131–154, 2002.

- [102] D. Spiliotopoulos, I. Androutsopoulos, C. D. Spyropoulos, "Human-robot interaction based on spoken natural language dialogue," 2001.
- [103] S. McGlashan, N. Fraser, N. Gilbert, E. Bilange, P. Heisterkamp, N. Youd, "Dialogue management for telephone information systems," in *Proc. of the Intl. Conference on Applied Language Processing, ICSLP'92*, 1992. [Online]. Available: [citeseer.ist.psu.edu/mcglashan92dialogue.html](http://citeseer.ist.psu.edu/mcglashan92dialogue.html)
- [104] IEEE Computer Society. IEEE Distributed Systems Online : Distributed Agents Projects. [Online]. Available: <http://dsonline.computer.org/agents/projects.htm>
- [105] P. Kiatisevi, V. Ampornaramveth, and H. Ueno, "A distributed architecture for knowledge-based interactive robots," in *Proceedings of the 2nd International Conference on Information Technology and Applications (ICITA 2004), Harbin, China*. Macquarie Scientific Publishing, 2004.
- [106] ECMAScript Language Specification, "http://www.ecma-international.org/publications/standards/ecma-262.htm." [Online]. Available: <http://www.ecma-international.org/publications/standards/ECMA-262.HTM>
- [107] The Mozilla Organization, "Rhino: JavaScript for Java." [Online]. Available: <http://www.mozilla.org/rhino/>
- [108] M. Hasanuzzaman, T. Zhang, V. Ampornaramveth, P. Kiatisevi, Y. Shirai, and H. Ueno, "Gesture based human-robot interaction using a frame based software platform," in *Proc. of the Intl. Conf. on Systems Man and Cybernetics (IEEE SMC 2004)*. IEEE, 2004.
- [109] T. Zhang, V. Ampornaramveth, P. Kiatisevi, M. Hasanuzzaman, and H. Ueno, "Knowledge-based multiple robots coordinative operation using software platform," in *Proc. of the 6th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2004)*. IEICE, 2004.
- [110] National Institute of Informatics (NII), Japan, "SPAK," June 2005. [Online]. Available: <http://sis.ex.nii.ac.jp/spak/>
- [111] H. A. Rowley, S. Baluja, and T. Kanade, "Neural network-based face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 20, number 1, pages 23-38, 1998.
- [112] M. A. Turk and A. P. Pentland, "Face recognition using eigenfaces," in *Proceedings of the 11th International Conference on Pattern Recognition*. IEEE, 1991, pp. 586-591.
- [113] University of Edinburgh, "The Festival Speech Synthesis System," June 2005. [Online]. Available: <http://www.cstr.ed.ac.uk/projects/festival/>



- [114] P. Kiatisevi, V. Ampornaramveth, H. Ueno, "Dialog Manager for Robots using Frame-based Knowledge Platform," in *Proc. of the 2nd International Workshop on Man-Machine Symbiotic Systems, Kyoto, 2004*.
- [115] H. Ueno and Y. Oomori, "Expert systems based on object model - an approach to deep knowledge systems," *Springer Lecture Notes in Engineering* 69, 1991.
- [116] N. Noy, R. Ferguson, and M. Musen, "The knowledge model of protege-2000: Combining interoperability and flexibility," in *Proceeding of 12th International Conference on Knowledge Engineering and Knowledge Management (EKAW)*. Springer, 2000.
- [117] M. Hewett, "Algernon - rule-based programming, <http://algernon-j.sf.net/>," June 2005. [Online]. Available: <http://algernon-j.sourceforge.net/>
- [118] J. F. Allen, "Time and time again: the many ways to represent time," *International Journal of Intelligent Systems*, vol. 6, pp. 341–355, 1991. [Online]. Available: [citeseer.ist.psu.edu/allen91time.html](http://citeseer.ist.psu.edu/allen91time.html)
- [119] Sony Corporation, "AIBO SDE, <http://openr.aibo.com/>," June 2005. [Online]. Available: <http://openr.aibo.com>
- [120] H. Bruyninckx, "The orocos project, <http://www.orocos.org/>," June 2005. [Online]. Available: <http://www.orocos.org/>
- [121] L. Erman, F. Hayes-Roth, V. Lesser, and D. Reddy, "The hearsay-ii speech understanding system: Integrating knowledge to resolve uncertainty," *Computing Surveys*, vol. 2, no. 12, pp. 213–253, 1980.
- [122] N. M. Sadeh, D. W. Hildum, T. J. Laliberty, J. McAnulty, D. Kjenstady, and A. Tsengy, "A blackboard architecture for integrating process planning and production scheduling," *Concurrent Engineering: Research and Applications*, vol. 6, no. 2, 1998.
- [123] M. Hans and W. Baum, "Concept of a hybrid architecture for care-o-bot," in *In proc. of ROMAN-2001*. IEEE, 2001, pp. 407–411.
- [124] J. Bos, E. Klein, and T. Oka, "Meaningful conversation with a mobile robot," in *Proc. of the 10th. Conf. of the European Chapter of the Association for Computational Linguistics (EACL)*. Association for Computational Linguistics, 2003.
- [125] Sandia National Laboratories. (2005, June) Jess: the rule engine for the java platform. [Online]. Available: <http://herzberg.ca.sandia.gov/jess/>

# Index

- backward chaining, 75, 79, 82
- communicative acts, 36
- CORBA, 26, 54
- DCOM, 26
- dialogue manager, 36, 91
- dialogue model, finite-state (state-based), 38, 103
- dialogue model, form-based, 39
- dialogue model, form-filling, 103
- dialogue model, information state, 40
- dialogue model, plan-based, 39
- dialogue system, 90
- dialogue systems, 35
- distributed systems, 25
- DSM (Distributed Shared Memory), 26
- Evaluator, 66, 79
- FIPA, 26, 54
- forward chaining, 75, 77
- frame, 19, 57
- frame systems, 28, 29
- frame, theory, 28
- HARIS robot system, 32, 33, 68
- Induce, 79
- JavaScript, 76, 88
- KFrame, 69, 88
- KFrameScript, 88
- Knowledge Manager, 52
- Knowledge Manager, roles, 58
- LSA, 95
- LSE, 95, 106
- message passing, 26
- NII, 15, 16
- OAA, 26
- OKBC, 30
- parser, 36
- platform approach, 18
- primitive agent, 49
- priority, 67
- reInduce, 79
- Rhino, 76
- RMI, 26
- robot, 22
- robot architecture, deliberative, 23
- robot architecture, hybrid, 25
- robot architecture, layered, 24
- robot architecture, reactive, 24
- robot, biologically inspired, 23
- robot, industrial, 22
- robot, socially interactive, 22
- robot, software, 22
- Robovie, 109
- RPC (Remote Procedure Call), 26
- Shakey robot, 24

SIS, 16  
slot, 29, 57, 69  
slot flags, 63  
SOAP, 26, 54  
SPAK, 19, 68, 114, 122  
special slots, 60  
speech recognition, 36  
SSS, 25  
Status Register, 93  
symbiosis, 16  
Symbiosis Information System (SIS), 15  
  
text-to-speech, 36, 112  
time-based layer, 65  
  
XML, 54, 71  
XML-RPC, 26, 54, 76  
  
ZERO++, 34, 68, 122

# Declaration

I, Pattara Kiatisevi, declare that this thesis is my original work and that, to the best of my knowledge, it contains no material previously published, or substantially overlapping with material submitted for the award of any other degree at any institution, except where due acknowledgment is made in the text.

Pattara Kiatisevi