Extendable Dialog Script Description Language for Natural Language User Interfaces

Kiyoshi Nitta
Yahoo Japan Research
Tokyo, Japan
Email: knitta@yahoo-corp.jp

***Abstract*—Natural language user interfaces are expected to be useful and provide a friendly atmosphere for human users working with intelligent human-computer interaction systems. Building correct and consistent large-scale script data for such interfaces requires large costs for construction and maintenance. Scripts written by many end-users in thoroughly flexible script languages might be a practical solution to this problem. We propose a dialog script description language and its framework. Two versions of script execution engines have been developed in C++ and Erlang programming languages. An example dialog script processed by one of these engines shows that a script editing function can be successfully represented by the script language. Because the script language has self-extending flexibility and ability to represent several versions of scripting languages, it increases the opportunity for end-users to accumulate and share a large amount of script data.**

***Index Terms*—natural language user interface; conversational agent; chat bot; chatter-bot; ontology extension**

I. INTRODUCTION

Constructing large-scale dialog script databases is one of the most critical issues for building practical natural language user interfaces that interact with users via typed text messages or voice utterances. Two major approaches can be considered to perform the task. The first approach is to apply machine learning technology to user operation or conversation logs for constructing dialog scripts. While the machine learning approach has an advantage in constructing large-scale scripts for reducing editorial operations of human staff, it might be difficult to achieve high precision adequateness of automatically generated scripts. The second approach is to accumulate dialog scripts created by many individual users. Script editing through natural language user interfaces might allow users to create their own dialog scripts. Because each user may have different preferences for editing scripts, such interfaces should be flexibly customized to maximize the opportunity.

With the second approach, we should address two technical challenges to operate a service in which many users safely customize natural language user interfaces for editing dialog scripts:

- dialog script languages and engines that are flexible for extending their description capability and modifying scripts by some types of script executions,
- load regulation methods for handling large-scale conversational transactions of many users.

We devised a knowledge representation framework [1] on which flexible dialog script languages can be represented. We developed several versions of flexible dialog script languages called *Graphical Language for Dialog Scripts (GLDS)* and two versions of GLDS execution engines coded in C++ and Erlang programming languages. In this paper, we explain a GLDS that address the first challenge by using the framework that can flexibly extend ontology. GLDS and its engines provide language elements that can modify operating dialog scripts. The Erlang version of the dialog script execution engine address the second challenge by using the parallel processing capability of Erlang language; however, this will be addressed in another study.

The main contributions of this paper can be summarized as follows:

- design of a flexible dialog script language (GLDS),
- implementation of GLDS execution engines, and
- evaluation of conversational operations for modifying dialog scripts.

The rest of this paper is organized as follows. Section II introduces the devised framework, execution engine algorithm, and GLDS dialog script language designed using the framework. Section III explains a self extendable example dialog script successfully executed using a system developed using the framework. Section IV describes the self-extending flexibility and ability of representing several versions of scripting languages. In Section V, several relevant studies on natural language user interface systems, graph-based knowledge representation methods, and semantic web technologies are summarized. This paper is concluded in Section VI.

II. GLDS FRAMEWORK

The framework described in this section consists of a data structure having basic semantics of dialog scripts and a dialog script execution engine that interprets the semantics. The data structure offers an extensible basis for GLDS. Dialog scripts executed by the engine are represented by *GLDS dialog scripts* based on this data structure.

A. Graph Structure and Semantics

The data structure of the framework is based on an augmented directed graph $ADG(E)$ [1] that permits edges to

 ISBN: 978-1-61208-269-1

connect to other edges as well as vertexes.

$$E_v = \{e(e_s, e_d) | e_s = \mathbf{null}, e_d = \mathbf{null}\} \quad (1)$$
$$E_{hoe} = \{e(e_s, e_d) | e_s \in E, e_d = \mathbf{null}\} \quad (2)$$
$$E_{toe} = \{e(e_s, e_d) | e_s = \mathbf{null}, e_d \in E\} \quad (3)$$
$$E_e = \{e(e_s, e_d) | e_s, e_d \in E\} \quad (4)$$
$$E = E_v \cup E_{hoe} \cup E_{toe} \cup E_e \quad (5)$$

Here, e means an *element*. Element e has a binomial structure (e_s, e_d), which is either an element of set E or an empty element **null**. An element of E_v is called a *vertex* and an element of E_e is called an *edge*. An element of $E_{hoe} \cup E_{toe}$ is called an *open edge*. Element set E is the union of sets of vertexes, edges, and open edges.

The semantics of the framework consists of three layers. The bottom layer semantics gives the meaning of the relationship to the edge set. If an element $e(e_s, e_d)$ is an edge, then e expresses the relationship from element e_s (source) to element e_d (destination). The ADG-based data structure permits element e_d to become an edge. That is, edge e can express the relationship to edge element e_d.

The middle layer semantics gives the class-instance structure by defining a vertex set $E_{apriori}$.

$$E_{apriori} = \{v_{class}, v_{ins}\} \subset E_v \quad (6)$$

The vertex v_{class} means that its instance elements are *classes*, and vertex v_{ins} means that its instance edges are *instance relations*. Consider the following element sets:

$$E_{ii} = \{e(v_{ins}, e_d) | \exists e_d \in E\} \quad (7)$$
$$E_{ins} = \{e | \exists e_{ii}(v_{ins}, e) \in E_{ii}\} \subset E_e \quad (8)$$
$$E_{class} = \{e | \exists e_i(v_{class}, e) \in E_{ins}\} \subset E_v \quad (9)$$

Element set E_{ii} includes all edges starting from the a priori vertex v_{ins}. Each element $e(v_{ins}, e_d) \in E_{ii}$ means that its destination element e_d is an instance relation. Edge set E_{ins} includes all these instance relation edges. Set E_{ins} is called an *instance relation set*. Each instance relation edge $e(e_c, e_d) \in E_{ins}$ means that its source vertex e_c is a class and that its destination e_d is an instance of the class. Vertex set E_{class} contains instance vertexes of the class v_{class} and is called a *class set*. Any set tuple (E_{ins}, E_{class}) should satisfy the following condition:

$$\forall e_i(e_c, e) \in E_{ins}, \exists e_c \in E_{class} \quad (10)$$

The top-level layer semantics defines the meaning of some vertex elements in class set E_{class}. Consider the following instance element set $\mathrm{ins}(e_c)$ for an arbitrary class vertex $e_c \in E_{class}$:

$$\mathrm{ins}(e_c) = \{e | \exists e_i(e_c, e) \in E_{ins}\} \subset E \quad (11)$$

Each element of set $\mathrm{ins}(e_c)$ inherits the meaning from class e_c. All functional definitions are combined to each element of class set E_{class}. Dialog scripts based on the framework are represented by the instance data of these classes. The data set is expressed as E_{data}:

$$E_{data} = \bigcup_{e_c \in E_{class}} \mathrm{ins}(e_c) \quad (12)$$

The data structure part of the framework is denoted as ADG_F:

$$ADG_F = (E_{apriori}, E_{ii}, E_{ins}, E_{class}, E_{data}) \quad (13)$$

In this graph structure, every element of the sets, except for $E_{apriori}$ and E_{ii}, has at least one edge that connects from a semantically defined vertex to the element. This characteristic ensures that the meanings of all elements can always be resolved by checking source nodes of incoming instance edges.

TABLE I
CLASS DEFINITIONS REQUIRED BY DIALOG SCRIPT TRACER (DST)

class	type	meaning
SRL	vertex	root of dialog script
SRK	vertex	matching keyword
REL	edge	transitional relation
RM	vertex	reply message
SEL	vertex	select action

B. Dialog Script Tracer

Dialog scripts are executed by a dialog controller called a *dialog script tracer* (DST). It requires the definitions of the classes in Table I. The 'class' column lists the mnemonic symbols of these classes, the 'type' column lists the vertex or edge type of the instance element of each class, and the 'meaning' column gives brief semantic explanations of these classes.

These classes are represented as class vertexes in the graph structure of the framework.

$$E_{class} = \{c_{srl}, c_{srk}, c_{rel}, c_{rm}, c_{sel}\} \quad (14)$$

Dialog scripts are expressed by instance elements $I_{srl}, I_{srk}, I_{rel}, I_{rm}$, and I_{sel}, where

$$I_{srl} = \mathrm{ins}(c_{srl}), I_{srk} = \mathrm{ins}(c_{srk}), I_{rel} = \mathrm{ins}(c_{rel}),$$
$$I_{rm} = \mathrm{ins}(c_{rm}), I_{sel} = \mathrm{ins}(c_{sel}) \quad (15)$$

There is no class of which an instance element can become both types.

$$I_{srl}, I_{srk}, I_{rm}, I_{sel} \subseteq E_v \cap E_{data} \quad (16)$$
$$I_{rel} \subseteq E_e \cap E_{data} \quad (17)$$

Only one instance element of the SRL class must be registered in a dialog script. When a new conversational session starts, the DST sets its context status to the vertex element. The DST can set its context status to one of the vertex set $V_{context}$ in an executing dialog script.

$$I_{srl} = \{root\} \quad (18)$$
$$V_{context} = I_{srl} \cup I_{sel} \quad (19)$$

 ISBN: 978-1-61208-269-1

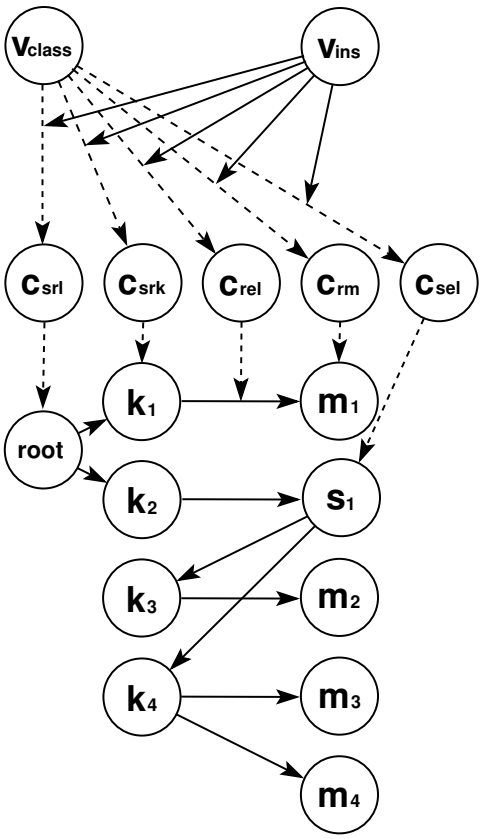

Fig. 1. Simple example of DST-consistent dialog script

At least one instance element of the SRK class must be connected to each element of $V_{context}$ using REL instance elements. When the context status is v_c and the DST receives a user input message, the DST attempts to find a v_c-related SRK instance element whose label matches the input message.

$$\forall v_c \in V_{context},\ \exists k \in I_{srk},\ \exists r(v_c, k) \in I_{rel} \tag{20}$$

At least one instance element of action type classes must be connected to each element of SRK.

$$\forall k \in I_{srk},\ \exists a \in V_{action},\ \exists r(k, a) \in I_{rel} \tag{21}$$

where

$$V_{action} = I_{rm} \cup I_{sel} \tag{22}$$

When an SRK instance element k is found to be a matched keyword, the DST randomly selects an action instance element a and executes it. If the element a is an instance of the RM class, the DST replies to the user with the text label of a and sets the context to $root$. If element a is an instance of the SEL class, the DST sets the context to a.

A simple example dialog script that can be executed by the DST is shown in Fig. 1. The dashed lines represent instance relations corresponding edge elements E_{ins}. Some are omitted for simplifying the diagram, e.g., relations between c_{srk} and k_2 or c_{rm} and m_2. In this example, $V_{context} = \{root, s_1\}$ and $V_{action} = \{s_1, m_1, m_2, m_3, m_4\}$. The diagram illustrates that the DST processes elements in I_{srk} and V_{action} alternatively.

TABLE II
SOME DEFINED CLASSES IN GLDS

	class	type	meaning
1	SM	edge	selection message relation
2	WCK	vertex	match a wild card keyword
	CFM	vertex	confirm an element
	CFA	vertex	confirm an array
	FMT	edge	format a relation
	NOP	vertex	no operation (dummy)
	SVC	vertex	save the context
	SLS	vertex	save the last selection
	SLR	vertex	save the last reply
	SVT	vertex	save a token
	QUI	vertex	quit
3	SUC	edge	success attribute
	FAI	edge	failure attribute
	IMM	edge	immediate attribute
4	VAR	edge	first argument
	VR2	edge	second argument
	VR3	edge	third argument
5	CTX	edge	variable (context)
	LSK	edge	variable (last selected keyword)
	LRP	edge	variable (last reply)
	MSG	edge	variable (message)
6	LET	vertex	set literal
	COP	vertex	copy
	ISN	vertex	check null
	CLR	vertex	clear
7	CSMS	vertex	search selection message
	CSMD	vertex	delete selection message
	CSMP	vertex	replace selection message

C. Graphical Language for Dialog Scripts

The meanings of dialog scripts are resolved through the meanings of all the elements in vertex set E_{class} of the framework data structure explained in Subsection II-A. We added class vertexes required by dialog scripts for describing practical conversational applications. Their meanings are written in program codes, which are almost separated from DST implementations. Their major classes are listed in TABLE II.

Classes are divided into seven categories: 1) those required to implement simple replies, 2) those accessing functions of the dialog controller, 3) those modifying transitional relations, 4) those specifying variable arguments, 5) those identifying variable entities, 6) those operating on variables, and 7) those accessing the data structure of the GLDS framework. Due to the space limitation of this paper, detailed description of each class is omitted. Some are explained in Section III.

III. EXAMPLE

GLDS dialog scripts can be modified through the execution of another GLDS dialog script that uses action instances to modify the data structure of the GLDS framework. An example of such a GLDS dialog script is shown in Fig. 2. The script modifies a selection message of an existing script. The selection messages are replied to the user when the DST change its context.

 ISBN: 978-1-61208-269-1

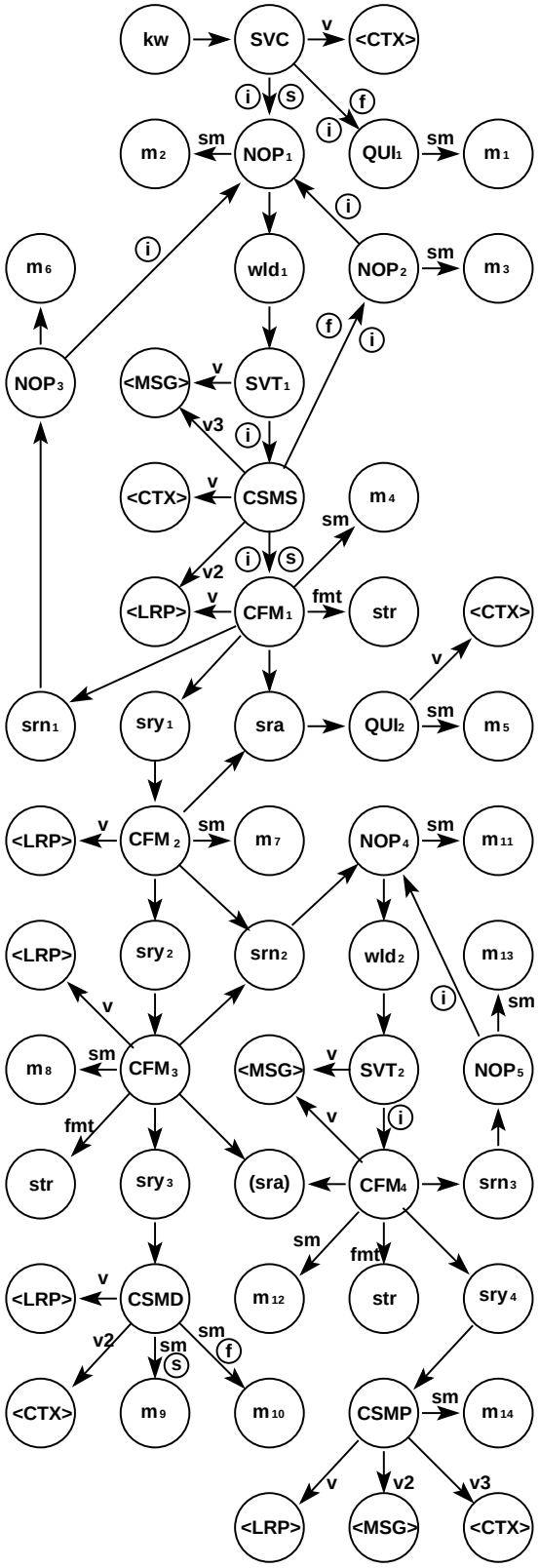

Fig. 2. Example GLDS dialog script for changing selection message

TABLE III
STRING DATA OF SCRIPT FOR CHANGING A SELECTION MESSAGE

symbol	class	string
kw	SRK	change selection message
str	RM	selection message: '%s'
sry_x	SRK	yes
srn_x	SRK	no
sra	SRK	abandon
m_1	RM	Move to any context.
m_2	RM	Specify a selection message to change.
m_3	RM	No matching selection message found.
m_4	RM	Do you want to change this selection message?
m_5	RM	Abandon changing the selection message.
m_6	RM	Specify a selection message again.
m_7	RM	Do you want to delete it?
m_8	RM	Really delete?
m_9	RM	Deleted.
m_{10}	RM	Could not delete the last selection message.
m_{11}	RM	Specify a new selection message.
m_{12}	RM	Do you want to replace the selection message with this?
m_{13}	RM	Specify a new selection message again.
m_{14}	RM	Replaced.

Instance edges and vertexes are represented by arrows and large circles in the diagram. A label consisting of capital letters in a circle indicates that the labeled vertex is an instance of the class whose name is indicated by the label. The number suffix of such a label identifies an instance from among those of the same class. A capital-letter label surrounded by angle brackets indicates that the vertex is a class vertex. This notation is used to represent variable references. A label consisting of lowercase letters in a circle indicates a vertex bound with string data (see TABLE III), except that the label "wld" indicates an instance vertex of the WCK class (see TABLE II), which is not bound with any string data.

String labels aligned with arrows indicate to which class each edge instance belongs. An arrow with no string label represents an instance edge of the transitional relation (REL) class. An arrow with the label "sm" represents an instance edge of the selection message relation (SM) class. An arrow with the label "v", "v2", or "v3" represents an instance edge of the argument variable relation (VAR, VR2, or VR3) class. An arrow with the label "fmt" represents an instance edge of the format relation (FMT) class. A small circle surrounding the characters "s", "f", or "i" indicates that the aligned edge has success, failure, or immediate attributes, respectively.

Fig. 3 shows an example of a dialog between a user (U) and the system (A) executing the script depicted in Fig. 2. It is assumed that the dialog began during the execution of another script. When kw string (see TABLE III) is detected, DST execution engine process SVC vertex that saves the current context pointer into variable <CTX>. Because this script

U:	Change selection message
A:	Specify a selection message to change.
U:	Source
A:	Do you want to change this selection message? selection message: 'source?' yes, no, abandon.
U:	Yes
A:	Do you want to delete it? yes, no, abandon.
U:	No
A:	Specify a new selection message.
U:	What source do you have in mind?
A:	Do you want to replace the selection message with this? selection message: 'What source do you have in mind?' yes, no, abandon.
U:	Yes
A:	Replaced.

Fig. 3. Example of dialog when changing selection message

was invoked during the execution of another script, the SVC process succeeds. DST follows the edge having immediate and success attributes, processes NOP_1 and m_2 that replies m_2 string to the user, and waits for user input. DST accepts any user input string, because wld_1 is connected to NOP_1 by a REL edge. Then DST proceeds to process SVT_1 that saves the user input string into variable <MSG>. Vertex CSMS searches a RM vertex that is connected from the vertex stored in variable <CTX> and has a text label matching the string stored in variable <MSG>. If the search task is succeeded, it stores the result into variable <LRP>. The explanations for the rest processes are omitted because of the space limitation of this paper.

IV. Discussion

GLDS has been developed by extending script description language specifications. We experimented with at least three internal versions: $GLDS_0$, $GLDS_1$, and $GLDS_2$. The basic finite-state-based dialog scripts are represented by $GLDS_0$, which contains the class vertexes in Equation (14). We extended $GLDS_0$ to $GLDS_1$ for representing dialog scripts that have script editing functions. $GLDS_1$ contains action classes, each of which performs a set of complex operations for a specific script editing task. $GLDS_1$ provides a safe environment in which inconsistent dialog scripts cannot be generated by any script editing operation. However, $GLDS_1$ must append class definitions with executable codes when a user requires new editing tasks. After such classes have been developed, we found that they can be represented by GLDS graphs connecting more reduced granularity of commonly reusable functions. We extended $GLDS_1$ to $GLDS_2$ by reducing complex classes to simple classes. Each $GLDS_1$-introduced class can be replaced by an adequate combination graph structure using $GLDS_2$-introduced classes. Some of these classes are listed in TABLE II. The 'changing a selection message' example explained in Section III and depicted in Fig. 2 is written in $GLDS_2$, which permits the representation of dialog scripts that may generate inconsistent dialog scripts due to their executions. This problem should be resolved in future studies.

The polymorphic framework holds executable script data written in these three GLDS versions. This makes it easier for the developer to try a new script description language even if the amount of accumulated legacy script data is huge. Our experimental system even permits the execution of dialog scripts that connect different versions of GLDS class instances. Although there is not enough space to show concrete examples in this paper, we can note that this benefit is derived by the graph structure design. As explained in Section II, the graph structure maintains an element set E_{class} that defines a kind of ontology giving the semantics of each dialog script element. The class set E_{class} can be extended by just adding classes that have new meanings, when a new version of GLDS is introduced. Therefore, no conflict occurs by resolving semantics of script elements, even if they are instances of different versions of GLDS classes.

The change selection message script example shown in Fig. 2 modified another GLDS dialog script through a conversation between the user and the system. It shows that GLDS and its framework provides a script modification function. We also implemented a GLDS dialog script that dynamically generates GLDS graph data and executes them. Although this was not efficient, it might be a solution to the problem of finite-state-based dialog control methods requiring a huge amount of scripts to achieve the conversational flexibility that can be provided using frame-based methods [2]. GLDS and its framework offer a kind of programming environment rather than a built-in customization language. Its specifications can increase in tandem with extensions of the system.

While the modification made by the change selection message script was applied to element set E_{data}, it is possible to modify element set E_{class}. If a class is created dynamically and its semantics defined, it means that the GLDS specifications are dynamically extended. A serious limitation of the current dialog script execution engines is that the semantics of classes is defined only by code written in a programming language (C++ or Erlang). Rich variations of E_{class} elements and a synthetic definition feature of an extended GLDS will overcome this restriction in the future. We hope to implement frame-based and agent-based dialog controllers [2] that cooperatively work with existing and extended DST/GLDS specifications. We believe that this infrastructure might help in the development of Minsky's emotional models [3] by using different types of dialog controllers and a polymorphic application of the knowledge model.

V. Related Work

Many natural language user interface systems have been developed. The earliest system, Weizenbaum's ELIZA [4], was reported in 1966. It was equipped with a reply function that repeated user messages with suitable pronouns substituted for

 ISBN: 978-1-61208-269-1

pronouns. To some degree, it gave users the impression they were conversing with a human psychiatrist. Colby's PARRY [5] acted as a psychiatric patient, and it was not required to reply with answers that were correct for its users. This showed that it is possible to implement a natural language user interface system that cannot be distinguished from a human psychiatric patient. The CONVERSE agent [5] integrated a commercial parser, query interface for accessing a parse tree, large-scale natural language resources like Wordnet, and other natural language processing technologies commonly available in 1997. It achieved a high percentage of completion, and won the 1997 annual medal of the Loebner prize [6]. Wallace's ALICE (http://alicebot.org/articles/wallace/dont.html [retrieved: February 2013]) promoted the XML-based script description language AIML. Wallace developed the AIML processor and the ALICE subset knowledge data in the AIML format as open source resources, and several chat bots [7], [8] have subsequently been developed based on these ALICE resources. ALICE has won the annual medal of the Loebner prize three times. While AIML provides one of the most customizable notations for natural language user interfaces, dialog scripts can only be authorized manually or externally. GLDS and its execution engines provide a dialog script notation, which has an ability to express the dialog scripts for authorizing other dialog scripts.

A semantic network is a well known graph structure having specific semantics. Griffith [9] defined it universally by dividing the net into a conveying net and an abstract net. The conveying semantic network corresponds to the whole data structure ADG_F that represents the expression system. The abstract semantic network corresponds to the element set E_{data} that represents the meanings of graph data. Griffith had no intention of extending the description capability of the abstract net, which is often the case, because semantic networks are used in natural language processing, where the semantics is considered to be stable. GLDS and its framework uses the net to represent dialog scripts, so we extended the net to be as extensible and flexible as an augmented semantic network [1].

Graph-based network data are commonly distributed using the resource description framework (RDF), which is a fundamental component of Semantic Web technology. In principle, ADG graph data can be expressed by the RDF. It is additionally required to provide a semantic basis like our GLDS framework to express GLDS script data using the RDF. The OWL Web Ontology Language can provide such a basis that partially represents the GLDS semantics. GLDS scripts expressed by OWL are convenient to port. It becomes more convenient to assume syntactical consistency if the scripts are expressed by OWL-DL, which is a subset of OWL, and can be processed by description logic (DL) [10] inference engines. Ontology extending issues are discussed in the linked data community [11] and large-scale ontology development [12]. Results of these studies might be used to improve the design of our framework.

VI. Conclusion and Future Work

To advance the research being done on natural language user interface systems, we have focused on methodologies that enable dialog script execution engines to preserve the consistency of their dialog scripts while using different types of dialog control methods simultaneously. We have also focused on technologies for implementing script editing functions by dialog scripts. We proposed and implemented a script description language (GLDS) and its framework that meet these requirements. We developed two versions of GLDS execution engines. A GLDS dialog script example was presented to show that a script editing function can be represented by $GLDS_2$ and executed successfully. Because GLDS and its engines have self-extending flexibility and ability to execute several versions of scripting languages, they increase the opportunity for end-users to accumulate and share a large amount of script data.

This framework has affinity for RDF-represented knowledge bases that increase their content rapidly via the semantic web and linked data activities. Integrating such knowledge bases and dialog scripts to increase the intelligence of natural language user interfaces is left for future work.

References

[1] K. Nitta, "Building an autopoietic knowledge structure for natural language conversational agents," in RuleML '08: Proceedings of the International Symposium on Rule Representation, Interchange and Reasoning on the Web. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 211–218.

[2] M. F. McTear, "Spoken dialogue technology: enabling the conversational user interface," ACM Comput. Surv., vol. 34, no. 1, 2002, pp. 90–169.

[3] M. Minsky, The Emotion Machine: Commonsense Thinking, Artificial Intelligence, and the Future of the Human Mind. Simon & Schuster, 2006.

[4] J. Weizenbaum, "ELIZA — a computer program for the study of natural language communication between man and machine," Commun. ACM, vol. 9, no. 1, 1966, pp. 36–45.

[5] Y. Wilks, Machine Conversations. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

[6] H. G. Loebner, "In response," Commun. ACM, vol. 37, no. 6, 1994, pp. 79–82.

[7] A. M. Galvao, F. A. Barros, A. M. M. Neves, and G. L. Ramalho, "Persona-aiml: An architecture developing chatterbots with personality," in AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems. Washington, DC, USA: IEEE Computer Society, 2004, pp. 1266–1267.

[8] G. O. Sing, K. W. Wong, C. C. Fung, and A. Depickere, "Towards a more natural and intelligent interface with embodied conversation agent," in CyberGames '06: Proceedings of the 2006 international conference on Game research and development. Murdoch University, Australia, Australia: Murdoch University, 2006, pp. 177–183.

[9] R. L. Griffith, "Three principles of representation for semantic networks," ACM Trans. Database Syst., vol. 7, no. 3, 1982, pp. 417–442.

[10] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds., The description logic handbook: theory, implementation, and applications. New York, NY, USA: Cambridge University Press, 2003.

[11] C. Bizer, T. Heath, K. Idehen, and T. Berners-Lee, "Linked data on the web (ldow2008)," in Proceedings of the 17th international conference on World Wide Web, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 1265–1266.

[12] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum, "Yago2: exploring and querying world knowledge in time, space, context, and many languages," in Proceedings of the 20th international conference companion on World wide web, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 229–232.

 ISBN: 978-1-61208-269-1