Hindawi Publishing Corporation
Journal of Robotics
Volume 2010, Article ID 301923, 14 pages
doi:10.1155/2010/301923

Research Article

An Extensible Dialogue Script for a Robot Based on Unification of State-Transition Models

Yosuke Matsusaka,[1] Hiroyuki Fujii,[2] and Isao Hara[1]

[1] *National Institute of Advanced Industrial Science and Technology (AIST), 1-1-1 Umezono, Tsukuba, Ibaraki, 305-8568, Japan*
[2] *Japan Science and Technology Agency (JST), 2-1-6 Sengen, Tsukuba, Ibaraki, 305-0047, Japan*

Correspondence should be addressed to Yosuke Matsusaka, yosuke.matsusaka@aist.go.jp

Received 1 November 2009; Revised 23 February 2010; Accepted 17 May 2010

Academic Editor: Noriyasu Homma

We propose extension-by-unification method to improve reusability of the dialogue components in the development of communication function of the robot. Compared to previous extension-by-connection method used in behavior-based communication robot developments, the extension-by-unification method has the ability to decompose the script into components. The decomposed components can be recomposed to build a new application easily. In this paper, first we, explain a reformulation we have applied to the conventional state-transition model. Second, we explain a set of algorithms to decompose, recompose, and detect the conflict of each component. Third, we explain a dialogue engine and a script management server we have developed. The script management server has a function to propose reusable components to the developer in real time by implementing the conflict detection algorithm. The dialogue engine SEAT (Speech Event-Action Translator) has flexible adapter mechanism to enable quick integration to robotic systems. We have confirmed that by the application of three robots, development efficiency has improved by 30%.

1. Introduction

In recent years, there has been an increasing demand for robots that work in a human life environment.

Replacement of human labor by robots in the manufacturing sectors (e.g., factory production lines) has already shown success. In the case of manufacturing robots, professional operators give commands to the robot. Professional operators have expert knowledge, and they are able to command the robot in a robot-friendly manner.

However, in the case of the robots used in a life environment, the operator who gives commands to the robot only has imperfect knowledge about the robot (called a "naïve user" hereafter). Naïve users often use natural language to command the robot. To create a robot that can be easily used by naïve users, the robot not only needs to have mechanical skills but also linguistic ability to understand a variety of commands.

The biggest problem in understanding language is diversity. Words used by a naïve user to command the robot will be diverse for various reasons (described in Section 3). This problem has been solved commonly by two methods: the machine learning methods and the behavior-based "scripting" methods. Each method has advantages and disadvantages.

An advantage of using the machine learning method is that the developer can implement the vast patterns of language understanding without any programming effort. For example, Iwahashi has used Markov model and stochastic context-free grammar to let the robot understand lexicons as well as associations between objects and words [1]. Roy has implemented on-line learning algorithm on a robotic platform, which automatically acquires the concept of the words and the objects [2]. However, the disadvantage of this method is that the models generated by the machine learning method cannot be edited or modified for reuse. Some methods enable retraining of the model by controlling a meta-level learning parameter (e.g., [3]), but we need to realize intended behaviors in complex situations, so it becomes generally difficult to find optimal learning parameters.

In contrast, in the case of scripting methods, the developer can program the specific behavior of the robot as intended. While the disadvantage is, however, the difficulty to cover the diversty of language understanding ability required in each application, because the effort of human developer is limited.

SHRDLU [4] is one of the most successful applications based on scripting method. The system was developed by Winograd in 1972. The system uses "inference-based" scripting approach. The script consists of planning part and vocabulary part and uses inference to complement the meaning of words.

The inference-based scripting approach is useful for the developer who has deep understanding about the inference system, but this requirement is sometimes difficult to fulfill in collaborative and incremental development (discussed later in Section 7.1).

Recently, "behavior-based" scripting method has been applied in many practical robotic systems. The application presented by Brooks [5] used hieratical structure model. The recent applications [6, 7] use state-transition model (finite state automata) to model the situation of the system. The developer incrementally develops the script by adding each behavior which fits to each small situation. Diverse situation understanding ability can be realized as a result of long-term incremental development.

The behavior-based scripting method can also be applied to communication robots by incorporating speech input with the situation model. Application of the behavior-based scripting method to the communication robot is first presented by Kanda et al. [8] in 2002. In their work, they not only proposed an incremental development framework, but also implemented an on-line development environment which can realize automated control of the robot. They have confirmed through a 25-day field study that with the help of the development environment, the conversation ability of the robot was incremented on line and succeeded to decrease the operation time of the human operator [9].

However, in the existing behavior-based scripting methods for communication robot, there is an inefficiency in terms of reusing the script to develop different types of robots (this problem is described in Section 3.1). In this paper, we present the extension-by-unification method in order to push forwards the behavior-based scripting approach to develop communication robots.

In our approach, we will not only focus on the ability of the model itself, but also on the descriptive format of the script and its operation. We show that the reuse can be enhanced by reformulating the conventional descriptive format and also show the effectiveness of the reformulation by implementing a computer-assisted development environment to enhance the development activity of the developer.

In Section 2, we give an overview of a basic state-transition model and its characteristics.

In Section 3, a formal discussion of incremental development methods for the state-transition model is presented. Here, we introduce the formalization of the proposed incremental development method and clarify its characteristics by comparing it to the previous method.

In Sections 4 and 5, the implementations of the script server and script engine are presented. The script engine and script server implemented support functions that will allow developers to reduce their development efforts.

In Section 6, examples of script development in actual applications are presented, and the effectiveness of the development environment is discussed.

2. State-Transition-Based Models

2.1. Formalization. A state-transition model is a modeling method in which the input and output of the system assume the following form:

$$A := \langle I, S, O, \gamma, \lambda, s_0 \rangle, \tag{1}$$

where I represents the input alphabet, O represents the output alphabet, S represents the internal states, γ represents the state-transition function, λ represents the output function, and s_0 is the initial state.

The state transition function γ is defined in association with the state to the input.

$$\gamma : S \times I \longrightarrow S. \tag{2}$$

The output function λ is defined in association with the state to the input.

$$\lambda : S \times I \longrightarrow O. \tag{3}$$

When the system is in state s_t and gets input alphabet i_t, state transition to s_{t+1} will occur as follows:

$$s_{t+1} = \gamma_{s_t, i_t}. \tag{4}$$

At the same time, we get output alphabet o_t as follows:

$$o_{t+1} = \lambda_{s_t, i_t}. \tag{5}$$

Even the input to the system is the same, the output of the system may be different, because the internal state s_t will be updated each time the system gets the input.

We have explained the state-transition model in an equation form, however, the state-transition model can be also presented in a 2-dimensional diagram called "state-transition diagram". In the diagram, each state is represented by a circle, and the transition between states is represented by arrows. In this paper, we annotate the transition conditions and the associative actions by including text over each arrow. We use a black circle (called a "token") to represent the current state.

For example, Figure 1 represents a conversation modeled by the state-transition model.

In the model presented in Figure 1, the initial state of the system is in "TV control" state. When the model gets the instruction "Turn on" as an input, it will output the command "turn-on-TV", and state transition "(a)" will occur. Then the token turns back to the same "TV control" state. When the model gets the instruction "Video" as an input, state transition "(b)" will occur, and the token will

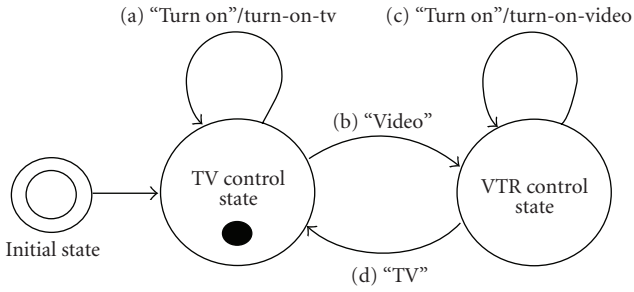


Figure 1: Example of state-transition model.

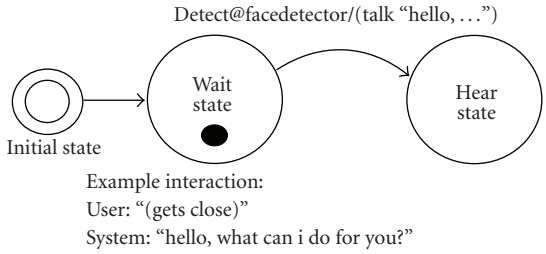


Figure 2: Example of state-transition model using multichannel input.

move to "VTR control" state. This time when the instruction "Turn on" is given, state transition "(c)" occurs and outputs the command "turn-on-video". In this way, we can model the context by defining an appropriate state and state transitions between the states.

The above example is expressed as follows in the equation form:

$$
\begin{aligned}
A &:= \langle I, S, O, \gamma, \lambda, s_0 \rangle, \\
I &= (\text{"Turnon"}, \text{"Turnoff"}, \text{"TV"}, \text{"Video"}), \\
S &= (\text{"tv-control"}, \text{"vtr-control"}), \\
O &= (\text{"turn-on-tv"}, \text{"turn-on-video"}, \\
&\qquad \text{"turn-off-tv"}, \text{"turn-off-video"}), \\
\gamma &= \begin{pmatrix} s_0 & s_0 & s_0 & s_1 \\ s_1 & s_1 & s_0 & s_1 \end{pmatrix}, \\
\lambda &= \begin{pmatrix} o_0 & o_2 & \text{none} & \text{none} \\ o_1 & o_3 & \text{none} & \text{none} \end{pmatrix}.
\end{aligned}
\tag{6}
$$

As we have seen here, the expression in equation form has an advantage in formalization, while the expression in diagram form has an advantage in quick understanding. In later discussion, we will use both the equation and the diagram forms to explain the concept quickly and formally.

State-transition model is a very simple get very powerful modeling method and has been applied to very wide applications. Because the structure of state-transition model is very simple, it is frequently misunderstood that the state-transition model can only model simple behavior. However, it can model diverse behavior by applying some extensions (e.g., [10, 11]).

2.2. Extensions

2.2.1. Multichannel Input. The original state-transition model uses a single input channel. In the case of a conversational system, the input channel is assigned to receive input from the speech recognition subsystem. However, it can accept multichannel input by formulating the transition function γ as $\gamma : S \times I \times C \rightarrow S$ and the output function λ as $\lambda : S \times I \times C \rightarrow \langle O, C \rangle$, where C is the type of input channel. By this extension, the model can integrate voice input as well as the other sensory inputs.

The example in Figure 2 shows the use of context in image and voice input.

2.2.2. Loop-Back Events. The state-transition model updates its internal state using external input. But by connecting output of the system to the input, it can realize autonomous behavior generation based on the internal event (in this paper, we call this a "loop-back event"). Loop-back events are important in realizing the autonomous behavior of the robot (examples are presented in Section 6).

2.2.3. Automatic Generation of Frame-Based Questions. A frame-based question is an interaction that requires answers to two or more questions in an arbitrary order. Example in Figure 3 shows realization of frame-based question using state-transition model. The structure of the model is apparently complex; however, we can generate this model using a simple algorithm.

2.3. Existing Implementations Used in Industry. There have been many script engines implemented (e.g., [12]). Most of them implement both multichannel and loop-back event extensions.

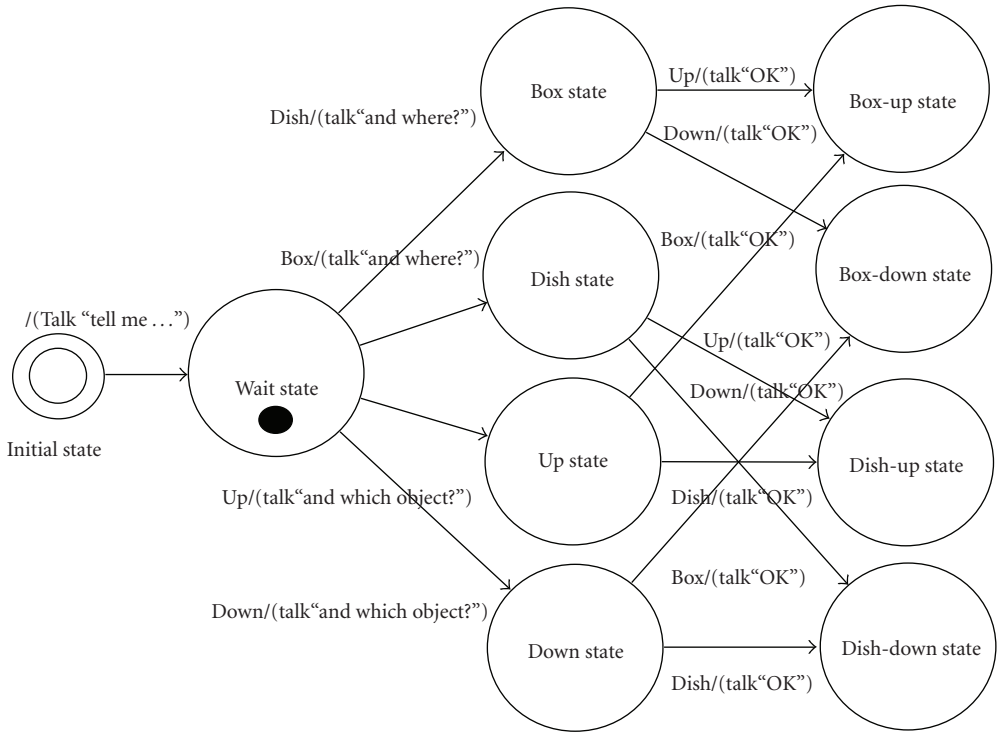
VoiceXML [13] is the de-facto standard of the script format used in various dialogue systems. It uses a more sophisticated format to describe the script than the state-transition model does. However, as we have shown the example of frame-based question in the previous section, we can easily convert the sophisticated description into the state-transition-based form. The implementation of VoiceXML script engines also uses this conversion, and the core part of these engines are based on a state-transition model.

Our script engine does not only implement the above extensions, but also has a function to support incremental development. In the next section, we discuss our incremental development method.

3. Incremental Development Method

Commands given by the human to the robot are diverse. The following are the factors that cause this diversity.

The Nature of Language. Human language is ambiguous, and different expressions can be used to give instructions that carry the same meaning.



Example interaction:
System: “tell me which object to move and which direction to move.”
User: “box.”
System: “and where?”
User: “up.”

```
(q1, q1n) = (("box", "dish"), "where")
(q2, q2n) = (("up", "down"), "which object")
func generate_2frame_question(q1, q1n, q2, q2n):
  for q1st, qn, q2nd in ((q1, q1n, q2), (q2, q2n, q1)):
    for quest1 in q1st:
      state[start].rule.push(quest1, "And "+qn+"?",
                                        state[quest1])
      for quest2 in q2nd:
        state[quest1].rule.push(quest2, "OK",
                                  state[quest1+quest2])
```

Figure 3: Example of state-transition model that can realize a 2-frame question. The structure of the model looks complex, but it can be generated easily using a simple algorithm.

Tasks. Robots working in a life environment have to accept a variety of tasks. In order to cope with this, it is necessary for them to understand a variety of commands.

Ability of the Robot Itself. The diversity is also caused by the ability of the robot itself. A command from a human becomes effective due to the functions of the robot. For example, humans do not say “walk N steps” to a robot on wheels.

The language comprehension system of the robot must be able to deal with these diversities.

In the script-based development approach, diversity has been dealt with by stacking a newly developed script onto the existing scripts. By accumulating a number of scripts, the developer can accumulate the number of commands that the system can deal with.

Incremental development of the state-transition model has previously been conducted using the “extension-by-connection” method (described in the next section). In

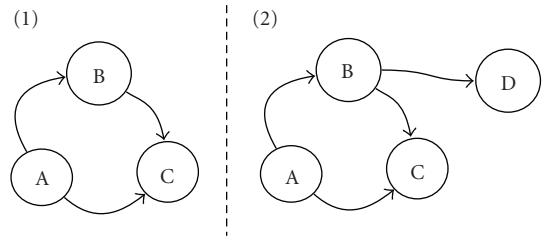


Figure 4: Extension of a state machine using the extension-by-connection model.

this section, we propose an "extension-by-unification" method that can cope with the diversities mentioned above (described in Section 3.3).

3.1. Extension-by-Connection Method. The simplest way to extend state-transition model is as follows.

(1) Add a new state to the existing state-transition model.

(2) Add a new transition from the existing state to the new state.

This process is illustrated in Figure 4.

Here, we formulate the above process. Let the existing state-transition model be A, and the accumulated state-transition model be A'.

As explained in Section 2.1, the existing state-transition model A can be represented by the following form.

$$A := \langle I, S, O, \gamma, \lambda, s_0 \rangle. \tag{7}$$

Here, S is the set of state $s \in S$. The transition function γ can be defined in any form. In this paper, we use the matrix of $S \times I$, in which the transition from state s_t to state s_{t+1} can occur if $\gamma_{S_t,i} = s_{t+1}$.

Similarly, we define the accumulated state-transition model A' as follows:

$$A' := \langle I, S', O, \gamma', \lambda', s_0' \rangle. \tag{8}$$

Then, the new state ΔS can be calculated as follows:

$$S' = S \cup \Delta S. \tag{9}$$

Here, $S' \cap \Delta S = \varnothing$.

The new state transition $\Delta\gamma$ can be calculated as follows:

$$\gamma'_{s_t,i} = \gamma_{s_t,i} \quad (s_t \in S, i \in I), \tag{10}$$

$$\gamma'_{s_t',i} = \Delta\gamma_{s_t',i} \quad (s_t' \in S', i \in I), \tag{11}$$

$$\lambda'_{s_t,i} = \lambda_{s_t,i} \quad (s_t \in S, i \in I), \tag{12}$$

$$\lambda'_{s_t',i} = \Delta\lambda_{s_t',i} \quad (s_t' \in S', i \in I). \tag{13}$$

The transition function of the accumulated part $\Delta\gamma$ needs to be defined based on the transition from the existing state S. Therefore, $\Delta\gamma$ will be a matrix of $S' \times I$. Note that the new state ΔS can be expressed only by the newly defined part, but the transition of the accumulated part $\Delta\gamma$ includes both old state S and new state ΔS in its definition.

The state-transition model is easy to understand in drawing a state-transition diagram. Extension-by-connection can also be carried out very easily by editing this diagram. There are several GUIs that can add state-transition rules through the operation of mouse clicks (e.g., [14]).

3.2. Problems with the Extension-by-Connection Method. Extension-by-connection is a useful method, but it has the following problems.

As we can see in (9) and (11), the definition of $\Delta\gamma'$ requires both S and ΔS. This causes problems in the function development of robots. For example, let us consider the following scenario.

(1) Robot "A" has function A, and we have already developed a state-transition model A^A to realize the function.

(2) For the robot "A" to accumulate function C, we have extended the state-transition model to A^{AC}.

(3) We have developed another robot, "B", which has function B. And we want to add function C to this robot.

Here, the state-transition model for function C is already developed for robot A. We want to reuse the model for robot B. Here, we discuss whether such a diversion would be possible.

First, the state S^{AC} is easily separable from state S^A and state S^C

$$S^C = S^{AC} - S^A. \tag{14}$$

However, the definition of state-transition function γ^{AC} is as follows:

$$\begin{aligned} \gamma^{AC}_{s_t^A,i} &= \gamma^A_{ij} \quad \left(s_t^A \in S^A, i \in I\right), \\ \gamma^{AC}_{s_t^{AC},i} &= \gamma^C_{ij} \quad \left(s_t^{AC} \in S^{AC}, i \in I\right). \end{aligned} \tag{15}$$

γ^C contains state S^A in its definition.

Because states S^A and S^B are defined for different types of robots, A and B are not equal. In addition, because the transition for the function C is defined dependently on state S^A, we cannot replace variables like $S^{AC} = S^{BC}$, which means that we cannot use γ^C to extend the state-transition model A^B. The state transition of function C developed for robot A cannot be diverted for the extension of robot B.

Ideally, once a feature is developed, it would be possible to share with other robots that need the same feature. In order to achieve this, we introduce the extension-by-unification method.

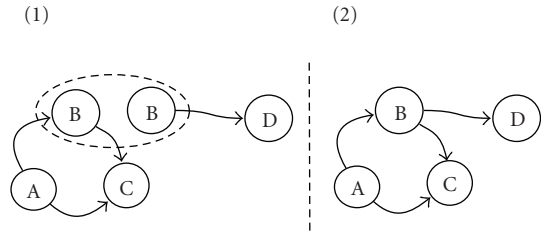


Figure 5: Extension of the state-transition model using the extension-by-unification method.

3.3. Extension-by-Unification Method. In the extension-by-unification method, we extend the state-transition model by the following procedure.

(1) Develop a state-transition model to realize a new function.

(2) Unify a state with the same ID between the existing and the new state-transition models.

This process is illustrated in Figure 5.

Here, we formulate the above process.

The existing state-transition model A can be represented by state S, state transition γ, and initial state s_0

$$A := \langle I, S, O, \gamma, \lambda, s_0 \rangle. \tag{16}$$

Similarly, the new state-transition model A' is represented as follows:

$$A' := \langle I, S', O, \gamma', \lambda', s_0' \rangle. \tag{17}$$

We accumulate the state-transition model A'' by unifying A and A'. First, we calculate state as follows:

$$S'' = S \cup S'. \tag{18}$$

Here, $S \cap S' \neq \emptyset$.

Next, the transition between the state S'' is calculated as follows:

$$\gamma''_{s_t,i} = \gamma_{s_t,i} \quad (s_t \in S, i \in I), \tag{19}$$

$$\gamma''_{s'_t,i} = \gamma'_{s'_t,i} \quad (s'_t \in S', i \in I), \tag{20}$$

$$\lambda''_{s_t,i} = \lambda_{s_t,i} \quad (s_t \in S, i \in I), \tag{21}$$

$$\lambda''_{s'_t,i} = \lambda'_{s'_t,i} \quad (s'_t \in S', i \in I). \tag{22}$$

By defining initial state s_0'' to be $s_0'' = s_0$, the extended state-transition model A'' will be as follows:

$$A'' = \langle I, S'', O, \gamma'', \lambda'', s_0'' \rangle. \tag{23}$$

As visible in (20), the transition function γ' is an $S' \times S'$ matrix that only includes state S' in its definition. The extension-by-unification method does not require the definition of the original state in the accumulated part of the state-transition model.

As noted in Section 3.2, in the conventional extension-by-connection method, the definition of the accumulated part of the state-transition model depends on information on the existing state. It is limited in terms of reusing scripts for this reason. The proposed extension-by-unification method does not have this problem. Using this method, we can significantly increase the reusability of the state-transition model (examples shown in Section 6).

3.4. Problems of the Extension-by-Unification Method. As discussed above, the extension-by-unification method can overcome a limitation in the extension-by-connection method by applying a simple reformulation. However, as a counterpart to this reformulation, we have dealt with the following problems that do not occur in conventional methods.

First, a conflict in transition conditions may occur. For example, when we try to unify two states with one another, the states may have different actions associated with the same transition conditions. In this case, the state-transition models cannot be unified.

Second, an isolated state may occur. For example, when we try to unify state-transition models that do not have the same state IDs in common, there will be no transitions between the old and the new states. In this case, the developer cannot activate the new function as intended.

In this study, we not only implement a script engine that has a state unification function (detailed in Section 5), but also implement a script-management server that includes conflict detection, isolated state detection, and executability detection functions (detailed in the next section).

4. The Development Environment

4.1. Script-Management Server. We developed the script-management system, which is based on wiki.

The developer can write the script in XML form on the wiki page, and the document of the script can also be written on the same wiki page. The developer can annotate each wiki page using tags. Tags are used as identifiers to indicate multiple pages working as a set.

Algorithm 1 is an example of the state-transition model written in the XML form.

Our run-time engine SEAT can read script using HTTP protocol. Thus, the developer can directly load and run the script (or the set of scripts defined by the tag) by specifying the URL.

The script-management server uses the core functions of dokuwiki (http://www.dokuwiki.org/). Functions described in the next sections are realized by extending the dokuwiki.

4.2. Detection of Isolated State. When we try to unify state-transition models that do not have state IDs in common, there will be no transition between the old and the new states. In this case, the developer cannot activate the new function as intended. Isolation of the state can be detected in Algorithm 2.

```
<state id="Robot" dict="julian-conf/hrp_operate">
 <rule>
   <key>[take] one step [forward]</key>
   <command host="talk">
      (talk "Take one step forward.")
   </command>
   <command host="control">
      (robot hwalk :set-target-pos 0.2 0 0)
   </command>
 </rule>
</state>
```

Algorithm 1

```
fstate = []

func checkisolatedstate_recur(stateid):
  for command in states(stateid).commands:
    if command.type == statetransition:
     if fstate(command.target) == 0:
       fstate(command.target) = 1
       checkisolatedstate_recur(command.target)

func checkisolatedstate(stateid):
  checkisolatedstate_recur(command.target)
  for state in states:
    if fstate(state) != 1
      detected = 1
```

Algorithm 2

4.3. Detection of State-Transition Conflict. When we try to unify two states with one another, the states may have different actions associated with the same transition conditions. In this case, the state-transition models cannot be unified. A conflict between the state-transition conditions can be detected in Algorithm 3.

4.4. Detection of an Unexecutable Action. An "unexecutable action" is an action that is defined in the state-transition model but cannot produce any output because the robot does not have the ability to generate the actual output. In this case, the developer cannot achieve the intended output. By using the instance ID of the adaptor mechanism (described in Section 5.2), an unexecutable action can be detected in Algorithm 4.

4.5. Visualization of Unifiable States. By using the above algorithms, the possibility of unification between scripts can be identified as "Unifiable", "Unifiable (occurrence of isolated state)", or "Conflict". Similarly, scripts can be classified as "Executable" or "Unexecutable". By comparing a script and an adaptor definition for the existing scripts, we can obtain a list of scripts annotated with 6 (3×2) classes.

Our script-management server displays the above list at the bottom of each wiki page. By displaying the list, the developer can easily find a script that can be included in his/her current application.

Figure 6 shows example of using the web-based interface.

5. Implementation of the Run-Time Engine

5.1. Architecture. SEAT consists of an adaptor mechanism, phrase matcher, automaton driver, and automaton unifier. In the next sections, we briefly overview each subsystem.

5.2. Adaptor Mechanism. The adaptor mechanism is used to connect the run-time engine to the other subsystems of the robot.

Adaptors are configured in XML format. For each adaptor configuration, an instance ID is defined. In the body of the state-transition model, the instance ID is used to describe the actions. By using this mechanism, even if the developer has changed the hardware configuration, the same state-transition model can be used by employing an adaptor definition that has the same instance IDs.

SEAT supports BSD socket communication, child process communication, UNIX standard input and output, and OpenRTM [15] as default interface types. Because the adapter mechanism is defined in an abstract form, the developer can easily add his/her own interface types.

```
func checkconflict(state1, state2):
  for c1 in state1.conditions:
    for c2 in state2.conditions:
      if (c1 == c2) && (c1.action == c2.action):
        detected = 1
```

Algorithm 3

```
func checkactions(adaptor, state):
  for c in state.conditions:
    if not exist c.action.instanceid in adaptor:
      detected = 1
```

Algorithm 4

5.3. Noise Robust Speech Recognition. A speech recognition function is also important in improving the accuracy of the robots' linguistic understanding. In the human life space, many noises occur around the robot. In such an environment, normal speech recognition algorithms are not accurate enough.

We have developed a speech recognition algorithm that works in a practical noise environment by using a signal processing technique combined with the speech recognition engine Julius [16]. Signal processing technique uses MUSIC spectrum method and fusion of video by Bayesian network, and it reduces environment noise by using ML beamforming (details are described in [17, 18]).

For HumanAID application (Figure 7), evaluation is done with two persons speaking simultaneously. Number of vocabulary was 492. Under this condition, the word error rate of speech recognition was over 19.9%, while the word error rate of normal speech recognition is 90.4%.

Speech recognition accuracy not only depends on environmental noises, but also depends on number of vocabularies. Because the recognizer needs to distinguish each word among given vocabularies, as the vocabulary increases, the recognition accuracy will go down. SEAT has a function to switch the speech recognition vocabularies depending on the situation. By using this function, the developer can increase the number of vocabularies of the total system while keeping the high speech recognition accuracy.

5.4. Phrase Matcher and Automaton Driver. The phrase matcher compares the input for each state-transition condition. To cope with the diversity of human language, we utilized a subset of regular expressions. If we write "[A]", phrase A is omissible. If we write "(A | B)", either phrase A or B can be matched.

When a match is found, the result is passed to the automaton driver. The automaton driver updates the current state and executes the commands based on the definition of the model. When a state transition occurs, switching of the speech recognition dictionary occurs at the same time.

6. Applications

6.1. Robots and Tasks. In this section, we present the applications we have developed using the development environment.

HRP-2. We have implemented the HumanAID task in the HRP-2 humanoid robot. The task is designed to assist people in everyday life. In the task, the robot greets the human, and the human gives commands to the robot, such as controlling the video or the TV, carrying drinks from the refrigerator to the table (Figure 7).

TAIZO. TAIZO is a health exercise demonstration robot [19]. It is a small robot character that greets people and demonstrates various exercises (Figure 8(a)).

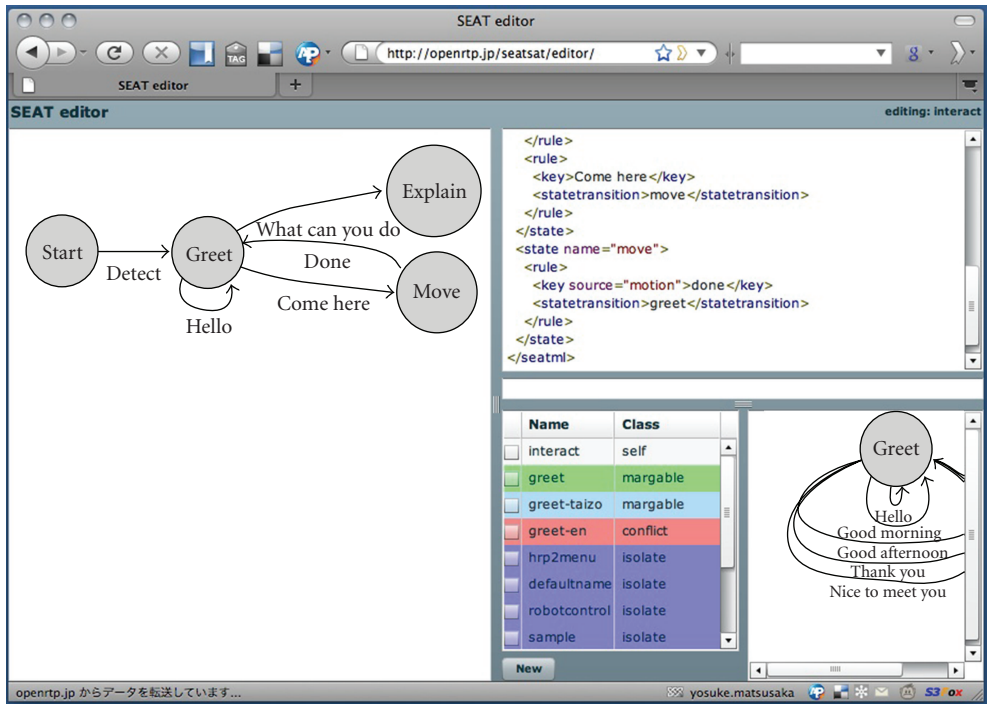
RH-1. RH-1 is a mobile robot that is designed to assist humans in the office environment (Figure 8(b)).

6.2. Development History. The development of the HumanAID task in HRP-2 has taken place from 2006 to 2007. Development of TAIZO and RH-1 has taken place from 2007 to the present. Table 1 shows the name of each script and its development period.

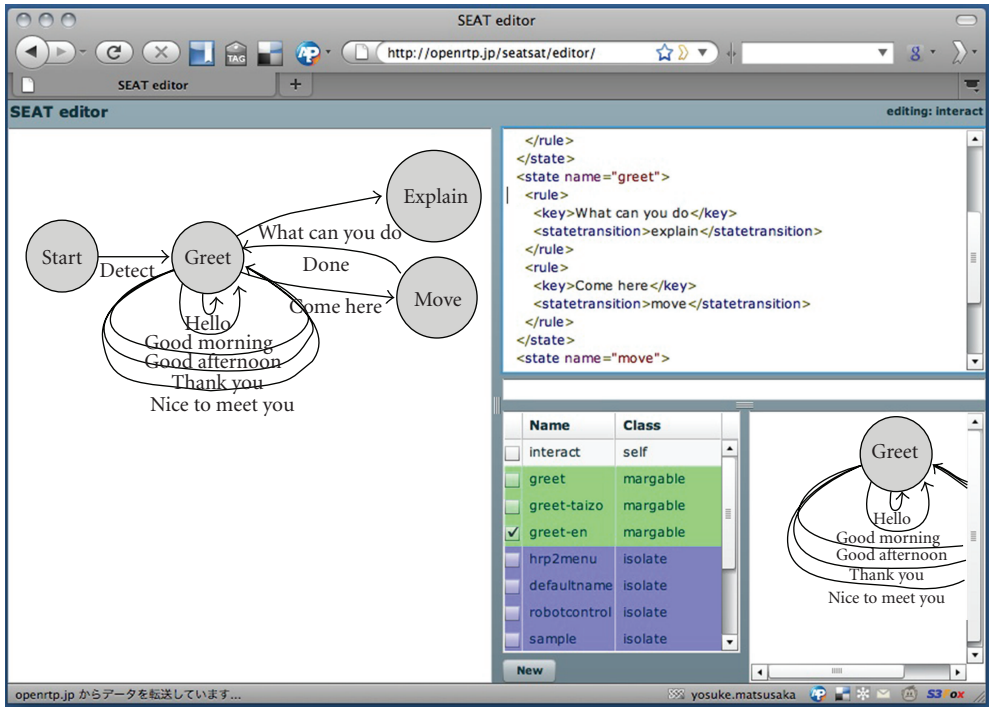
Here, we list the development history.

Unit-Based Development in HRP-2 (Period 1). HumanAID task functions have been developed separately. The state-transition model for demo conversation (e.g., saying "hello", "bye", introducing itself), the model for controlling the robot (e.g., walking, picking up objects), and the model for controlling the TV and VTR using an infrared controller were split into different scripts and developed simultaneously in this period.

Integration in HRP-2 (Period 2). After the development of each part of the function, a script was developed that



(a)



(b)

FIGURE 6: Example of using the web-based interface. (a) Overview of the development interface. Visualization of the state-transition model (left), XML-based editing panel (right top), real-time annotation of existing scripts (right bottom). Editing task script. When the developer types the keyword "Hello", the existing script from the script database is annotated as "conflict" and suggests reuse. At this step, the system only accepts 3 ("Hello", "What can you do?", "Come here") phrases. (b) When the developer checks the "greet" script, which already contains several vocabularies for greeting, it is unified to the task script. As a result, the developer only has to increment the application specific vocabulary to realize the whole script with many vocabularies. At this step, the system accepts 7 ("Hello", "Good morning", "Good afternoon", "Thank you", "Nice to meet you", "What can you do?", "Come here") phrases.

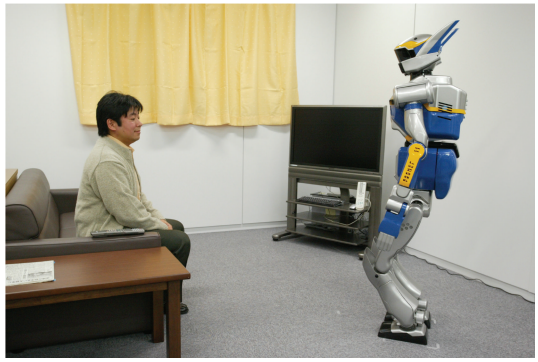
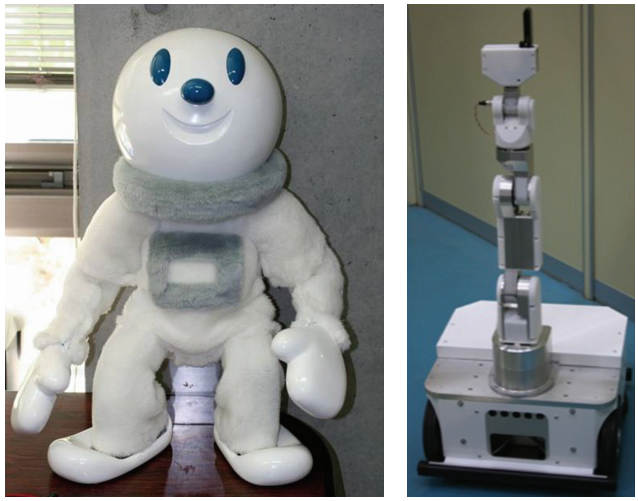


Figure 7: HRP-2 during a HumanAID task.



(a) (b)

Figure 8: TAIZO (a) and RH-1 (b) robots.

defined a menu. Within this script, a central state and states corresponding to each function were defined. For states corresponding to each function, nothing was contained in this script, but it was unified with other scripts to obtain the functions. Only transitions from the central state to each function state were defined in the script.

Each automaton has been developed and tested individually, but at the final stage of development, it was possible to unify the script by simply confirming the warning messages given by the script-management server.

Extension of the Scripts in TAIZO (Periods 3 and 4). Script development of TAIZO was conducted by extending the scripts developed for HRP-2. The scripts for greeting and basic demo tasks were selected for reuse, and the other scripts (TV-control, VTR-control) were not selected because TAIZO has no ability to control this equipment. To add more patterns to the greeting, the "greet-taizo" script was defined, which shares the same state as "greet" but adds more transitions. Functions specific to TAIZO were developed as the script "exercise". Finally, a menu script was developed to integrate all of the functions.

Table 1: Scripts used by each application and its development period.

Name of script	Used by			# of trans.	Period (developed for)
	HRP-2	TAIZO	RH-1		
greet	∗	∗	∗	3	1, 3 (HRP-2)
robot-ctl	∗	∗		13	1 (HRP-2)
tv-ctl	∗		∗	14	1 (HRP-2)
vtr-ctl	∗		∗	10	1 (HRP-2)
hrp-menu	∗			3	2 (HRP-2)
greet-taizo		∗		4	3 (TAIZO)
exercize		∗		17	3 (TAIZO)
taizo-menu		∗		17	4 (TAIZO)
wander			∗	15	5 (RH-1)
ask-who			∗	3	6 (RH-1)

Although the composition of the subsystems (e.g., speech recognizer, behavior generation) of TAIZO and HRP-2 was different, it was possible to share the scripts with no modification by simply switching the adaptor configurations.

Development of RH-1 (Periods 5 and 6). The script development of RH-1 was conducted simultaneously with the development of TAIZO. In RH-1, some control functions were imported from HRP-2. The script "wander" was defined as wandering around the office. This script not only uses speech input, but also uses visual information to find people. A multichannel input mechanism was used to integrate visual and speech inputs.

6.3. Results and Effects of Development. As a result of these developments, the number of acceptable command types has reached 43, 54, 45 for the respective applications.

Each application shares 93%, 30%, or 60% of its scripts with the other applications, respectively. As a total, 30% ($= 1 - (3+13+14+10+3+4+17+17+15+3)/(43+54+45)$) of the script development effort is reduced. Because the time used to develop the system was in proportion to the number of transitions defined in the script, the development time is estimated to have decreased by 30%.

7. Discussion

7.1. Comparison with Other Accumulative Development Methods. In the above discussion, we compared our method to the extension-by-connection method. Both the extension-by-connection and extension-by-unification methods belong to the same state-transition model group. For other modeling methods, and especially for artificial intelligence applications, a production system model is used in some applications.

A production system model is a modeling method that maintains the state of the system as a multidimensional feature vector, and controls the execution of actions by comparing the state to the pattern written in the script.

By using the same symbols as in Section 2, the production system model is formalized as follows:

$$A := \langle \mathbf{S}, R, s_0 \rangle, \tag{24}$$

where $\mathbf{S}$ is the state in vector (in the state-transition model, S was a set of states), R represents the conditions for each rule, and $\mathbf{s}_0$ is the initial state in vector.

Condition R is a comparison function that can be defined arbitrarily. In this paper, we define R as a matrix that consists of the conditions for each rule R_{ij}. The rows of R stand for each rule, and columns stand for the conditions corresponding to each dimension of features in the state vector. We define the conditions as "1": the value of the feature is positive, "−1": the value of the feature is negative, and "0": does not care.

Production system models are generally known to be extensible. Our model can easily accumulate rules, as follows:

$$\begin{aligned} R''_{i,f_m(j)} &= R_{i,j} \quad \left(R_{i,j} \neq 0, i, j \in S\right), \\ R''_{i+N,f'_m(j)} &= R'_{i,j} \quad \left(R'_{i,j} \neq 0, i, j \in S'\right), \\ R''_{i,j} &= 0 \quad (\text{otherwise}, i, j \in S''), \end{aligned} \tag{25}$$

where R'' represents the accumulated rules, R represents the existing rules, and R' is the rule for accumulation. f_m is a mapping function that converts each feature vector dimension into the other. S is a dimension of the feature vector, and N is the number of rules.

Although these are good points theoretically, in practical development there have only been a few examples of successful large-scale development. It is generally said that in a production system, the developers are required to be proficient in script development in order to ensure the extensibility of the system. We discuss this problem from the viewpoint of handling the states.

Let developer "A" has defined the state $\mathbf{S}_A$ and rule R_A, and developer "B" has accumulated rule R_B to extend the system. Rule R_B not only extends the rule, but also adds a new dimension to the state vector that is not used in rule R_A. Let the new state vector be $\mathbf{S}_B$ and the old state vector be $\mathbf{S}_A$. The final state vector in the extended system is $\mathbf{S}_{A+B} = \mathbf{S}_A \cup \mathbf{S}_B$.

In the extended system, the rules R_A and R_B are evaluated on an equal footing. However, when developer "A" developed rule R_A, only $\mathbf{S}_A$ was considered. On the other hand, when developer "B" developed, rule R_B, $\mathbf{S}_{A+B}$ was considered. $\mathbf{S}_{A+B}$ contains more information than $\mathbf{S}_A$, and this may cause an antinomy to rule R_A, which only considers $\mathbf{S}_A$. Figure 9 illustrates this antinomy.

To avoid this problem, the script developer needs to project the final system before beginning to develop the rule R_A, and maintain consistency during the development of rule R_B. However, this problem is as difficult as the frame problem [20] discussed in early artificial intelligence research.

In contrast, the state-transition model clearly defines the model in the form of state and transition conditions in the design phase.

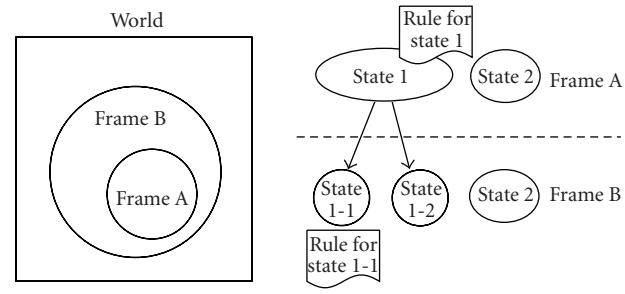


Figure 9: Antinomy between an existing rule and an accumulated rule. State 1 may be split into state 1-1 and state 1-2 when a new feature vector (frame B) is considered. There are no clear criteria to use to decide whether to select rule 1 or rule 1-1.

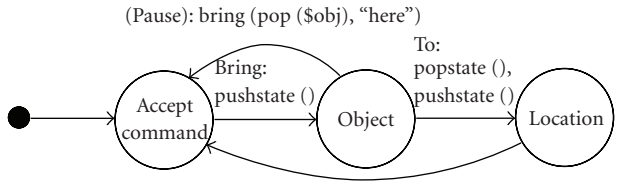
In state-transition model, we need to define a new state every time we add a new feature vector to the internal state so that the system can handle it. Here, we explain this using a concrete example. In rule R_A, developer "A" has considered feature A and defined $S_1 = \langle A = 0 \rangle, S_2 = \langle A = 1 \rangle$. Developer "$B$" wants to consider feature B in addition to feature A. Here, developer "B" has to define the new states $S_3 = \langle A = 0, B = 0 \rangle, \ldots, S_6 = \langle A = 1, B = 1 \rangle$. In the production system model, the state with smaller dimensions will be included in the state with larger dimensions. In the above case, all states will be included in $S = \langle A, B \rangle (A, B \in [0, 1])$, and states S_1 and S_2 will be overwritten.

In the state-transition model, the developer assesses the internal parameters of the system in the design step, but for the description, the developer needs to break down the combination of parameters into a set of states. When the developer wants to increase the internal parameters, he/she has to use a different state. Due to this restriction, the definition of a state is always clear, and is not overwritten by a script that is added later. Thus, the developer can proceed without being trapped by the issues discussed earlier in this section.

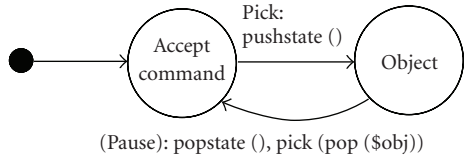
7.2. Reuse of Motion Content. In this paper, we have proposed a development environment to enhance the reuse of dialogue components. However, total development cost of the robot has to be calculated from both speech communication part and motion generation part.

Because the cost for reusing the motion depends on the algorithm, we first explain the algorithm we used. There are two methods for robot motion generation, one is the planning-based algorithm, and the other is the motion database. In our examples, for HumanAID and RH-1, we have used the planning-based algorithm. For TAIZO robot, we have used the motion database.

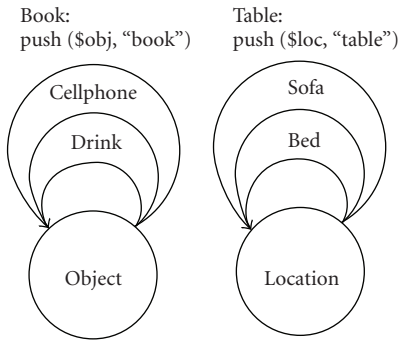
In the planning-based algorithm (HumanAID and RH-1) the motion generation algorithm will automatically generate the motion. We do not need to adjust the motion by hand, but only have to change the parameter such as structure of the arm, location of the target object. Because we share the motion generation algorithms between the robot, the cost of reusing the motion is very low in this case.



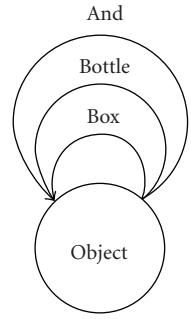
(a) Syntactics model for “bring” command



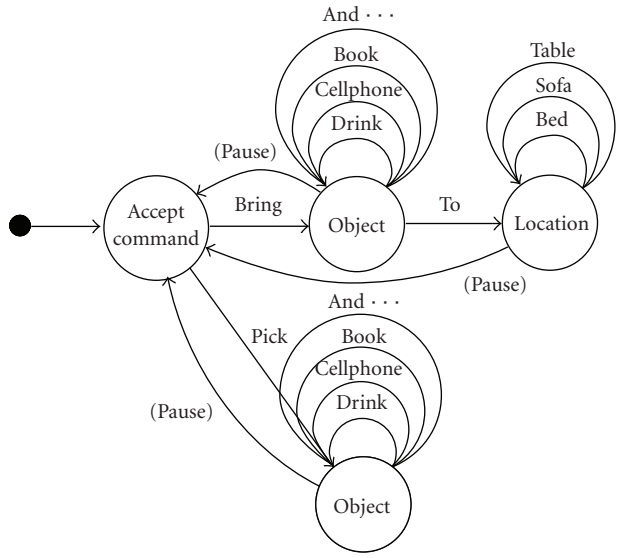
(b) Syntactics model for “pick” command



(c) Vocaburary for objects and locations (part of command is abbriviated)



(d) vocaburary for additional objects



(e) Unified automaton (commands are abbriviated in this figure)

Figure 10: An example of word-level input model.

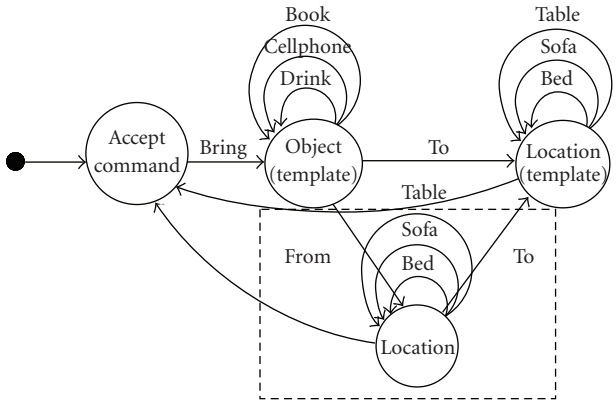


Figure 11: An example of three automaton unification. Dotted rectangle part is the extended part from the example shown in Figure 10.

When we use motion database (TAIZO), there is a problem in reusing the motion. Because the motion database is generated by hand, we have to create a new motion by hand for the new robot with different structure. For this problem, we are planning to implement motion retargeting technique. This algorithm was originally developed for creating a motion for computer animated creatures in movies [21]. The algorithm automatically generates a motion of animated creature from the motion of human actor by calculating the mapping between both structures. By using this motion retargeting algorithms, we believe the cost of reuse for motion database-based robot can be also reduced to very low level.

7.3. Word-Level Input and Pushdown Functions. State-transition model is possible to model the input in word level. An example is shown in Figure 10.

In the example, first, state-transition model in Figures 10(a), 10(b), and 10(c) is developed to interpret command. Second, state-transition model in Figure 10(d) is developed to extend the vocabulary. Finally, all the developed models are unified as Figure 10(e). This model uses pushdown automata [22] ("pushstate" and "popstate" command) to enable transition to the shared state "object" and return back. It also uses "push" and "pop" commands to hold the information about each object and location. Now the system can accept the input such as: "Bring Cellphone (pause)" converted as bring (cellphone, here), and "Bring Drink to Sofa (pause)" converted as bring (drink, sofa).

In our application, we have used the input in command (sentence) level. Modeling of input in word-level may be also useful, especially when the developer wants to share the syntactical structure of commands among the several scripts.

7.4. Comparison with the Template Methods. In VoiceXML, the industrial standard for scripting general voice operation system, there is an extension called RDC (Reusable Dialogue Component) [23]. The VoiceXML-RDC allows the developer to write the scripts of primitive interactions in an abstract form. The user can instantiate the primitive interactions by filling in the template parameters. By using the template, the user can generate dialogue scripts which fit to their application with less programming effort.

Our proposed method has equivalence to the template method. As we have seen in Figure 10, "object" slot of the syntax can be filled either with Figures 10(c) or 10(d).

The difference between our method and the template method is the possiblity to compose single model like in Figure 11 because our method uses "state" as a unit of unification, and allows unification of two or more components to one,. The template method is not able to realize this example, because it explicitly distinguish template part and instance parameters and only allows to unify those two.

7.5. Left Problems and Future Research. As shown in the examples of previous section, by using the proposed extension-by-unification method, the developer can develop functions simultaneously, and in the final step he/she can easily unify those functions to create an integrated system. In addition, scripts created in the past can easily be reused in the new application. In the conventional extension-by-connection method, the developer had to develop each function in turn, because it did not support the "merging" of scripts that had been developed simultaneously. Moreover, the developer needs to erase the unneeded functions manually when he/she wants to reuse the script in another robot. The proposed method does not require this process, because the script keeps information about the function even after the integration, and it can easily be separated for reuse. In our example, the script created for HRP-2 could be reused for TAIZO, but we had to remove the functions for TV and VTR control because TAIZO does not have this capability. In the conventional extension-by-connection method, we would have to do this manually. However, in the extension-by-unification method, this process is done simply by selecting the scripts to be unified.

As a result, we have confirmed that the extension-by-unification method significantly improves the efficiency of developing conversational function of the robots.

In terms of unification problems, it was possible to prevent the occurrence of isolated states by displaying a warning message, but there was a problem when the developer intentionally isolated the state. This happens when the developer has written a script in a redundant manner, or when he/she has tried to use pushdown automation. We are currently trying to solve this problem using 2 methods. One is a more intelligent isolation detection algorithm that can reduce misdetection (e.g., [24]), and the other will allow the developer to use an annotation tool that indicates that the state is intentionally isolated.

By applying a probabilistic weight to each transition of the state-transition model, the model became equivalent to the Markov model. There are some robots that have realized interactions with humans using such a model (e.g., [25]). In this paper, we have discussed the accumulation of the state-transition model based on deterministic input and output. However, probabilistic models are effective in modeling

real-world information, which includes noise. In further research, we are planning to incorporate these probabilistic models to the extensible development environment we have developed in this paper.

8. Summary

In this paper, we have proposed an extension-by-unification method to improve reusability and flexibility in the incremental development of state-transition models. The dialogue engine SEAT has been developed to realize the incremental development of state-transition models to give robots a dialogue ability that can cope with various kinds of speech inputs in various tasks. SEAT has a flexible adaptor mechanism that can connect to many types of robotic interfaces, and the developer can accumulate scripts by using the script server, which has a function to propose existing reusable scripts to the developer. We have confirmed that the application of this system to the development of three robots has significantly improved the efficiency of their development.

References

[1] N. Iwahashi, "Language acquisition through a human-robot interface," in *Proceedings of the International Conference on Spoken Language Processing (ICSLP '00)*, vol. 3, pp. 442–447, Beijing, China, 2000.

[2] D. Roy, "Grounded spoken language acquisition: experiments in word learning," *IEEE Transactions on Multimedia*, vol. 5, no. 2, pp. 197–209, 2003.

[3] A. L. Symeonidis, I. N. Athanasiadis, and P. A. Mitkas, "A retraining methodology for enhancing agent intelligence," *Knowledge-Based Systems*, vol. 20, no. 4, pp. 388–396, 2007.

[4] T. Winograd, *Understanding Natural Language*, Academic Press, New York, NY, USA, 1972.

[5] R. A. Brooks, "Intelligence without representation," *Artificial Intelligence*, vol. 47, no. 1–3, pp. 139–159, 1991.

[6] L. Kaelbling, "A situated-automata approach to the design of embedded agents," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 85–88, 1991.

[7] B. Yartsev, G. Korneev, A. Shalyto, and V. Kotov, "Automata-based programming of the reactive multi-agent control systems," in *Proceedings of the IEEE International conference on Integration of Knowledge Intensive Multi-Agent Systems (KIMAS '05)*, pp. 449–453, Waltham, Mass, USA, 2005.

[8] 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*, pp. 1848–1855, Anchorage, Alaska, USA, May 2002.

[9] T. Kanda, M. Shiomi, Z. Miyashita, H. Ishiguro, and N. Hagita, "An affective guide robot in a shopping mall," in *Proceedings of the ACM/IEEE International Conference on Human-Robot Interaction*, pp. 173–180, La Jolla, Calif, USA, 2009.

[10] F. Huang, J. Yang, and A. Waibel, "Dialogue management for multimodal user registration," in *Proceedings of the International Conference on Spoken Language Processing*, vol. 3, pp. 37–40, Beijing, China, 2000.

[11] M. Denecke, "Informational characterization of dialogue states," in *Proceedings of the International Conference on Spoken Language Processing*, vol. 2, pp. 114–117, Beijing, China, 2000.

[12] SMC, "The State Machine Compiler," http://smc.sourceforge.net/.

[13] "Voice Extensible Markup Language (VoiceXML) Version 2.0," http://www.w3.org/TR/voicexml20/.

[14] "NEC RoboStudio," http://www.necst.co.jp/product/robot/.

[15] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W. Yoon, "RT-middleware: distributed component middleware for RT (robot technology)," in *Proceedings of theIEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3555–3560, Ibaraki, Japan, 2005.

[16] T. Kawahara, A. Lee, T. Kobayashi, et al., "Free software toolkit for Japanese large vocabulary continuous speech recognition," in *Proceedings of the 6th International Conference on Spoken Language Processing (ICSLP '00)*, vol. 4, pp. 476–479, Beijing, China, 2000.

[17] I. Hara, F. Asano, H. Asoh et al., "Robust speech interface based on audio and video information fusion for humanoid HRP-2," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '04)*, pp. 2404–2410, Sendai, Japan, October 2004.

[18] F. Asano, K. Yamamoto, I. Hara et al., "Detection and separation of speech event using audio and video information fusion and its application to robust speech interface," *EURASIP Journal on Applied Signal Processing*, vol. 2004, no. 11, pp. 1727–1738, 2004.

[19] Y. Matsusaka, H. Fujii, T. Okano, and I. Hara, "Health exercise demonstration robot TAIZO and effects of using voice command in robot-human collaborative demonstration," in *Proceedings of the IEEE/RSJ International International Symposium on Robot and Human Interactive Communication*, pp. 472–477, Toyama, Japan, 2009.

[20] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," *Machine Intelligence*, vol. 4, pp. 463–502, 1969.

[21] M. Gleicher, "Retargetting motion to new characters," in *Proceedings of the Annual Conference on Computer Graphics and Interactive Techniques*, pp. 33–42, ACM, Orlando, FLa, USA, July 1998.

[22] M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing, Boston, Mass, USA, 1997.

[23] "Reusable Dialog Components (RDC) Tag Library," http://jakarta.apache.org/taglibs/doc/rdc-doc/.

[24] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: application to model-checking," in *Proceedings of the 8th International Conference on Concurrency Theory*, pp. 135–150, Warsaw, Poland, 1997.

[25] H. Asoh, Y. Motomura, I. Hara, S. Akaho, S. Hayamizu, and T. Matsui, "Combining probabilistic map and dialog for robust life-long office navigation," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 807–818, Maui, Hawaii, USA, 1996.