



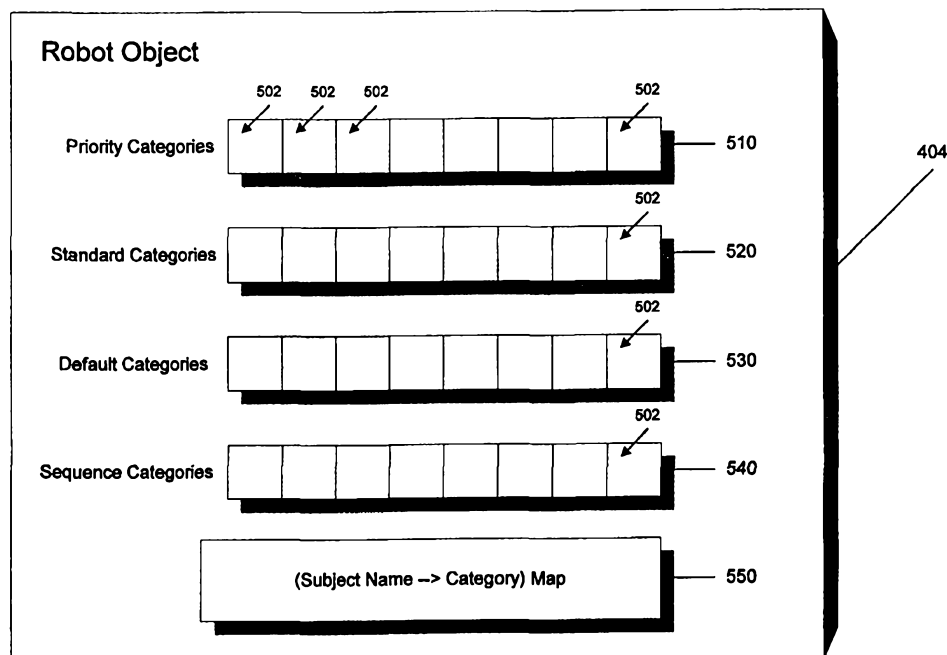
INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁶ : G05B 11/00, H04M 1/64		A1	(11) International Publication Number: WO 98/55903
			(43) International Publication Date: 10 December 1998 (10.12.98)
(21) International Application Number: PCT/US98/10536		(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).	
(22) International Filing Date: 22 May 1998 (22.05.98)			
(30) Priority Data: 08/868,713 4 June 1997 (04.06.97) US 09/017,760 3 February 1998 (03.02.98) US 09/018,213 3 February 1998 (03.02.98) US 09/067,322 27 April 1998 (27.04.98) US			
(71) Applicant (for all designated States except US): NEUROME-DIA, INC. [US/US]; 298 4th Avenue #400, San Francisco, CA 94118 (US).		Published <i>With international search report.</i> <i>Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.</i>	
(72) Inventors; and (75) Inventors/Applicants (for US only): TACKETT, Walter, A. [US/US]; 216 4th Avenue #12, San Francisco, CA 94118 (US). BENSON, Scott, S. [US/US]; 1050 Carolan Avenue #214, Burlingame, CA 94010 (US). DILLINGER, Ray, S. [US/US]; 106 Santa Maria Drive, Novato, CA 94947 (US).			
(74) Agent: KALER, Stuart, P.; 298 4th Avenue #400, San Francisco, CA 94118 (US).			

(54) Title: VIRTUAL ROBOT CONVERSING WITH USERS IN NATURAL LANGUAGE**(57) Abstract**

A system is herein provided comprising mechanisms (114, 100, 110, 106, 107, 108, 200, 300) and methods of defining, creating, operating, and testing virtual robots ("BOT") (404). A virtual robot (404) as provided herein encompasses categories (510, 520, 530, 540, 550) and topics (710, 720, 730, 740, 750, 402, 820, 830, 840, 630) of natural conversation that can be encoded in a natural "script language" (108, 118, 120). During the course of operation, these categories (510, 520, 530, 540, 550) are activated according to the conversation (900 - 3002) taking place with a user. Different categories (510, 520, 530, 540, 550) are focused at different times

according to changes in the conversation (900 - 3002). The system provides both automatic and explicit mechanisms (114, 100, 110, 106, 107, 108, 200, 300) for focusing categories (510, 520, 530, 540, 550) and topics (710, 720, 730, 740, 750, 402, 820, 830, 840, 630) during the course of conversation (900 - 3002). Additionally, if more than one category (510, 520, 530, 540, 550) seems appropriate to answer a particular user statement, the system provides mechanisms (114, 100, 110, 106, 107, 108, 200, 300) for selecting the most appropriate response (900 - 3002). Similarly, the system provides mechanisms (114, 100, 110, 106, 107, 108, 200, 300) for verifying the accuracy and appropriateness of the virtual robot's response (900 - 3002) by encoding test examples into the virtual robot script (108, 118, 120).



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece	ML	Mali	TR	Turkey
BG	Bulgaria	HU	Hungary	MN	Mongolia	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MR	Mauritania	UA	Ukraine
BR	Brazil	IL	Israel	MW	Malawi	UG	Uganda
BY	Belarus	IS	Iceland	MX	Mexico	US	United States of America
CA	Canada	IT	Italy	NE	Niger	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NL	Netherlands	VN	Viet Nam
CG	Congo	KE	Kenya	NO	Norway	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NZ	New Zealand	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	PL	Poland		
CM	Cameroon	KR	Republic of Korea	PT	Portugal		
CN	China	KZ	Kazakstan	RO	Romania		
CU	Cuba	LC	Saint Lucia	RU	Russian Federation		
CZ	Czech Republic	LI	Liechtenstein	SD	Sudan		
DE	Germany	LK	Sri Lanka	SE	Sweden		
DK	Denmark	LR	Liberia	SG	Singapore		
EE	Estonia						

VIRTUAL ROBOT CONVERSING WITH USERS IN NATURAL LANGUAGE

Technical Field

The present invention relates to virtual robots and, in particular, to virtual robots
5 conversing with users in a natural language.

Background Art

Understanding and processing natural language, as either spoken or written by humans,
has long been a goal in the field of artificial intelligence. As computers have been programmed
10 of late to perform awe-inspiring feats, such as defeating the world's best human chess master in
his game, other skills exhibited by humans are still seemingly beyond the reach of even the most
powerful computers. Although a small child may not be able to play chess, that child typically
has the facility to process and understand its native tongue. Computers, on the other hand, have
yet to exhibit any significant level of mastery in the realm of natural language processing.

15 Understanding natural language requires that the recipient have both (1) linguistic
knowledge of the particular language; and (2) sufficient world knowledge to carry on a
meaningful discourse about a certain topic. One early attempt at simulating natural language
skills included a program called ELIZA. To simply model human language skills, ELIZA
matched input sentences from a human against simple input templates of keywords. An input
20 template might, for example, take a group of related keywords and lump them together for the
purposes of responding. Thus, words like "father, mother, brother, sister" might be grouped
together for a response that relied on the concept of "family". Thus, with ELIZA, familiar words
were recognized while the rest of the sentence was ignored and a response was generated based
on the matched keywords.

25 Although the method for processing natural language was greatly simplified by the
approach taken by ELIZA, there were several problem areas that needed to be addressed by this
approach. One such problem area occurs in the situation where more than one input template
matched a given input sentence. ELIZA solved this problem by associating a "rank number" for
each keyword. ELIZA would then select the highest ranked word in the sentence for matching.
30 This mechanism for resolving multiple template matches, although very easy to implement, is
relatively unsophisticated in adequately capturing the semantic meaning of natural discourse.

For the purposes of the present invention, "virtual robots" (or "BOTs") are software
programs that interact and/or communicate with users (human, machine or otherwise) that take
actions or make responses according to input from these users. More sophisticated BOTs have

since been developed to capture a deeper sense of a particular discourse's meaning. As these BOTs became more sophisticated, more processing work needed to be accomplished to capture an adequate world knowledge (albeit about a limited range of topics) and linguistic knowledge.

One of the major drawbacks is that early BOTs were written in very specific computer languages, such as C, PASCAL and the like. Thus, in order to create BOTs, its creator would need to author in this programming language. Additionally, modification of such BOTs were just as cumbersome -- any change to a BOT would need to be made to source code in a complex programming language to be subsequently recompiled.

One early solution to the ease of creation and modification included the use of "scripting" languages that allowed for the creation and modification of BOTs in a language more akin to English. In fact, ELIZA itself used "scripts" to implement input sentence template matching.

Another problem with very early BOT creation tools was managing the complexity of English language interaction. For example, there were no models in ELIZA for the state of an on-going conversation. Thus, during the course of a conversation, a human user logged on to a BOT might talking about a "mouse" in one thread of conversation where the mouse is of the mammalian type. Then, during another thread of conversation, the particular user could refer to a "mouse" of a computer peripheral type. These early BOTs found it difficult to handle the semantic shift of word meaning.

Another example is the "pronoun" problem. During the course of a lengthy conversation, a human user can use the pronoun "it" to refer to an object that has long been discussed, but not immediately. Humans typically look for other semantic clues in the conversation to correctly bind the word "it" to the object meant. Early BOTs lacked adequate features to decipher these semantic clues to glean the correct meaning.

One early attempt to resolve these problems is described in an article by Michael Mauldin entitled "CHATTERBOTs, TINYMUDs, and the Turing Test: Entering the Loebner Prize Competition". Mauldin describes "activation networks" that are networks of response nodes that are connected to other responses through weighted edges. Prior to operation, an activation network would be created or instantiated by defining a list of subject-response nodes, whereby any given subject that may be discussed by a human user would elicit a given response as defined by a node. Different, but related, subject-response nodes would be connected via edges having a pre-defined weight. Subject-response nodes would themselves carries an activation numeric value that would vary during the course of operation of the BOT. As the subject discussed by the human user matches a given subject of a node, that node's activation value

would increase. Similarly, the activation value of adjacent subject-response node to that first node might also increase or decrease according to the weight given their connecting edge.

The main drawback with activation networks is their difficulty in creating and maintaining them. For example, in designing the initial network, it is difficult to accurately
5 assign any meaningful activation values for particular nodes and the weights accorded to their edges. Additionally, it is difficult to debug such systems because any adjustment of the weights and values will have unpredictable effects later. The problem is that the numerical assignment of natural language topics is a hard task to accomplish with meaningful results.

Thus, there is a need in the art to have a means for easily creating virtual BOTs that
10 naturally focuses the attention of the BOT to resolve inherent ambiguities of natural language that humans do so well at -- without the need to assign somewhat arbitrary numerical values that have unforeseen or unintended consequences and interactions.

There is also a need to have attention-focusing mechanisms that are easily extensible and modifiable without the need for a major re-coding effort in a traditional computer programming
15 language.

There is also a need for a virtual robot to have an automatic learning feature that allows it to learn from its interactions with users and improve its own behavior.

In another aspect, if the BOT script allows more than one response to match an input, and does not provide any explicit total ordering on the possible responses, there are several ways in
20 which the set of possible responses could be used. The BOT execution system might present all of the possible responses to the user, either in ordered or unordered form, or it might select one, or several, of the possible responses. In the case where the BOT execution system is either ordering the possible responses or selecting some subset of the responses, the execution system must have an automated way of deciding among the set of possible responses.

Therefore, in the case where a BOT script provides conditions under which a response is
25 appropriate, there is a need in the art for an automatic method of selecting a response from a number of possible responses such that the BOT author does not need to make the conditions mutually exclusive nor impose a fixed ordering on the conditions.

There is also a need for this mechanism to produce appropriate responses to both context-
30 sensitive and context-independent inputs. For example, on an input such as "Who is he?", the context of such an input is not entirely obvious. But, other inputs such as "Who is the president of Neuromedia?", should elicit a unique response that is less dependent on the context of the statement. Thus, this mechanism should attempt to work equally well on both types of

statements. In particular, there is a need to handle the problem of ambiguity that the pronouns inject into a conversation.

In another aspect of BOT interaction, automatic pronoun replacement is very effective for the pronouns “he”, “she”, “him”, and “her”, but does not work effectively in practice for other pronouns such as “it”, “they”, “that”, and “you”, due to the frequency with which these words are used in senses other than as pronouns. The focus of attention stack is generally successful at distinguishing among multiple answers to a single question (such as “what is it?”) but lacks any effective way of deciding whether such an answer is preferable to a default response such as “I don’t know.” For instance, suppose there is one topic that answers the question “where is it?” if the user has been discussing Neuromedia, but no topic that answers the question “where is it?” when “it” refers to San Francisco. In this case, even if the user has been discussing San Francisco and has not mentioned Neuromedia at all, if the user then asks “what is it?”, they may get a response describing the location of Neuromedia rather than the correct response, which in this case is probably “I don’t know.” Given that a BOT’s knowledge is in practice always incomplete, such situations can repeatedly arise during the interaction of a user with a BOT.

Thus, there is a need in the art to have a method of selecting a response to a context-sensitive question or statement, and for that method to allow a question to be answered with a default answer or no answer at all, even when the identical question would get a specific answer if asked in another context.

In another aspect, if a BOT is to be deployed in a publicly accessible way such as a web page or chat site, there is a need to test the BOT as thoroughly as possible to ensure that, as often as possible, it will produce an appropriate response to the inputs that it is likely to receive and the situations that it is likely to encounter. In this context, “input” refers to any description of a situation the BOT may encounter; although the most common inputs are textual inputs from users, inputs can be actions taken by users, external circumstances, or even events internal to the BOT such as an internal alarm clock. If the BOT can be tested in advance, the person or organization that is deploying the BOT can be more certain of its likely performance, and errors can be detected in advance that might otherwise result in mistakes that could mislead users interacting with the BOT and/or reflect poorly on the authors or deployers of the BOT.

Therefore, there is a need in the art to have a means of automatically verifying the performance of a BOT that allows the creation of the testing information simultaneously with the development of the BOT and that allows the BOT author to easily modify the testing information as the BOT is modified.

There is also a need for the verification mechanism to be given sufficient information to provide useful diagnostic output when an error is found, in addition to simply reporting the error.

There is also a need, in the case where the response given by the BOT may vary depending on previous inputs given to the BOT, for the verification mechanism to be able to verify that a response will be given correctly regardless of the prior sequence of inputs the BOT has seen, or that a response will be given correctly under the condition that a particular sequence of inputs precedes it. There is a need for such verification to be done efficiently, without the need for testing an exponential number of sequences of inputs and responses.

Disclosure of Invention

The present invention meets these aforementioned needs by providing manners and methods of defining, creating, operating, and testing virtual robots. A virtual robot as provided according to the principles of the present invention encompasses categories and topics of natural conversation that can be encoded in a natural "script language". During the course of operation, these categories are activated according to the conversation taking place with a user. Different categories are focused at different times according to changes in the conversation. The present invention provides both automatic and explicit mechanisms for focusing categories and topics during the course of conversation. Additionally, if more than one category seems appropriate to answer a particular user statement, the present invention provides mechanisms for selecting the most appropriate response. Similarly, the present invention provides mechanisms for verifying the accuracy and appropriateness of the virtual robot's response by encoding test examples into the virtual robot script.

Brief Description of Drawings

Figure 1 depicts a suitable operating environment for the purposes of the present invention.

Figure 2 depicts different operating environments for the purposes of the present invention wherein the connection to user 116 is via an intranet or internet connection.

Figure 3 depicts yet another operating environment wherein the BOT processor is merely a client of another server, such as a chat room or virtual world server.

Figure 4 expands the view of one embodiment of the runtime executive suitable for the purposes of the present invention.

Figure 5 expands the view of an embodiment of the robot object suitable for the purposes of the present invention.

Figure 6 describes the content of a category as embodied in a robot object.

Figure 7 expands the view of user record as shown in **Figure 4**.

Figure 8 expands the view of local variables as found in **Figure 4**.

5 **Figure 9** shows a view of one embodiment of a program object that is used to select a category given a user input, suitable for the purposes of the present invention.

Figure 10 expands the view of the structure corresponding to a single condition-action block in the BOT script, as used in **Figure 9**.

10 **Figure 11** expands the view of the structure corresponding to a single low-level condition in a BOT script, as used in **Figure 10**, suitable for use in a category selection system such as that shown in **Figure 9**.

Figure 12 shows a view of one embodiment of a compound condition, suitable for use in a category selection system such as that shown in **Figure 9**.

Figure 13 expands the view of a tester for user properties shown in **Figure 9**.

15 **Figure 14** shows a view of one embodiment of a node in a pattern-matching structure, suitable for efficient parallel pattern-matching in a category selection system such as that shown in **Figure 9**.

Figure 15 shows a data structure created from a particular BOT script for pattern-matching in a category selection system.

20 **Figure 16** shows a hierarchy of conditions and low-level blocks created from a particular BOT script for use in condition evaluation in a category selection system.

Figure 17 expands the view of one part of the data structure shown in **Figure 15**, showing details that were simplified in **Figure 15**.

Figure 18 depicts one view of an overall process of the use of the verification mechanism.

25 **Figure 19** depicts the overall behavior of the verification mechanism.

Figure 20 depicts the detailed behavior of the verification mechanism when applied to a particular input that is to be verified.

Figure 21 depicts the detailed behavior of the verification mechanism when applied to a special case of conditions that are to be verified in a particular sequence

30 **Figure 22** depicts the structure that is created for the verification of a particular sequence of conditions discussed below.

Figure 23 shows the architecture for automatic learning of features in the context of the present embodiment of the present invention.

Figure 24 shows the feature extractor of the learning mechanism.

Figure 25 shows the learner module of the learning mechanism.

Figure 26 shows the report module of the learning mechanism.

Figure 27 shows the code generation module of the learning mechanism.

Figure 28 shows the process of feature extraction in greater detail, following the processing of one raw data sample.

Figure 29 shows the way of computing conditional probabilities of labels on features.

Figure 30 depicts the data flow of the learning process.

Best Mode for Carrying Out the Invention

I. OVERVIEW AND GENERAL ARCHITECTURE

The term "robot" is used interchangeably with "BOT" throughout the remainder of this application. For the purposes of the present invention, both "BOT" and "robot" refer to any program which interacts with a user in some fashion, and should not be assumed to refer only to physically embodied robots.

Referring now to **Figure 1**, the operating environment of the present invention is depicted. The environment can be characterized generally into three partitions: front end 102; BOT processor 100; and back end 104. Front end 102 is generally the environment in which a human user 116 consults a virtual BOT interface 114 via a computer 112 that may be connected to the BOT processor via a communications link, such as through a server connected to the Internet or alternatively directly connected to BOT processor 100. It will be appreciated that many other means of connection to BOT processor 100 are well known to those skilled in the art and that the present invention should not be limited to the any particular aspects of the general operating environment as disclosed herein.

Typically, human user 116 connects to a site whose interface of first impression is a virtual BOT interface 114. The advantage for the site developer is that human user 116 may have a help or information request that is easily handled via BOT interface 114. Today, it is not uncommon to find sites having a list of FAQs ("Frequently Asked Questions") that serve this purpose of handling very low level user concerns and questions. However, for more advanced questions or interactions with the site, virtual BOTs will become increasingly popular.

In the operating environment of this embodiment of the present invention, BOT interface 114 is an instantiation of a process that is spawned by BOT processor 100 via connection 110. BOT processor 100 itself may comprise connection 110; runtime executive process 106, compiler 107, and a set of BOT programs 108. As users 116 log onto a site having BOT

processor 100 via connection 110, runtime executive 106 executes an interaction routine that guides the discussion that occurs between user 116 and BOT processor 100. Typically, a two way communications dialogue occurs between user 116 and BOT processor 100 wherein user 116 may ask questions, make declarative statements and other normal communications patterns that humans typify. For the purposes of the present invention, "communications" is to be very broadly interpreted. Indeed, suitable communications could be in the form of written or spoken language, graphics, URL's or the like that may be passed to and from a user to an automatic interface program, such as the present invention.

In turn, runtime executive 106 parses the statements and questions generated by the user and responds according to a set of BOT programs 108. As will be discussed in greater detail, BOT programs 108 are typically created at the back end 104 as a set of "scripts" that the BOT processor will tend to engage in with user 116. For example, if the site using BOT processor 100 is a site for a reseller of personal computers, then BOT processor 100 should be designed to handle questions and discussions concerning personal computers and their peripherals in general. Thus, the back end 104 will generate scripts that will guide the discussion concerning many computer-related topics. These script programs 108 are then compiled by compiler 107 and the compiled code is incorporated into runtime executive 106. As will be discussed below, these scripts are written in an English-like language called "Gerbil™" -- the name derived from "General Robot Building Language", as developed by the present assignee, Neuromedia, Inc..

As the two-way discussions between user 116 and runtime executive 106 continue, it is generally desirable to engage in quality control of BOT processor 100. This quality control is provided at back end 104 via feedback loop comprising a transcript of dialogues 118 and backtrace and state information 120 of the BOT processor 100; a supervisor 122 and editor 124. As transcripts develop over the course of interacting with a user, the text of these transcripts are stored, together with the state of the runtime executive and backtrace of execution through the runtime executive code. This information forms the basis for accurately diagnosing the runtime executive and for debugging its performance. Such information may be stored electronically in a storage media or could be printed out in human readable form.

Supervisor 122 analyzes the information at 118 and 120 with an eye towards optimizing the performance of the runtime executive. Typically, supervisor 122 could be another human, deciding if the semantics captured by the system needs to be upgraded in response to a dialog transcript that has occurred. If so, supervisor 122 could optionally invoke an editor 124 to edit the Gerbil programs that represent the semantic framework of the runtime executive. These programs would then be re-compiled and incorporated into the runtime executive. Supervisor

122 could be a software program (as opposed to another human) that would automatically seek to analyze the performance of the runtime executive and make corrections to the runtime executive through the editing process.

5 **Figures 2 and 3** depict slightly different operating environments for the purposes of the present invention. **Figure 2** depicts a situation wherein the BOT processor 200 connects to user 116 is via an intranet or internet connection e.g. web connector 210. For example, web connector 210 may thus spawn a Java applet 216 inside of an HTML page 214 to provide the two-way communications as discussed above. It will be appreciated that such use of Java applets embedded in HTML pages is well known to those skilled in the art. Alternatively, HTML page 10 214 might communicate directly with web connector 210 via a CGI connection or some other well-known connection protocol. Likewise, the BOT server can accept HTML requests directly. In such cases, persistent state information can be tracked by a "cookie" set in the web browser or other means.

As is shown, supervisor 122 interfaces with robot executive 106 via console process 212. 15 Console process 212 monitors the execution of robot executive 106 and may do so with a reduced set of monitoring tasks, such as merely checking to see if robot executive 106 is actively running. **Figure 3** depicts a situation wherein the BOT processor 300 is merely a client of another server, such as a chat room or virtual world server, as depicted by server 320. BOT processor 300 is connected to server 320 via a chat/virtual world connector 310 in much the 20 same fashion as any client would connect to a server site. Users 116 that desire to talk or converse with the BOT processor 300 interact through their client connections 314 in much the same fashion as any client-to-client communications that would be effected on server 320.

Although **Figures 1, 2 and 3** give a general description of various operating environments in which virtual BOTs may exist, it will be appreciated that many other operating 25 environments are obvious to those skilled in the art and that the scope of the present invention should not be so limited to the exemplary descriptions as given above.

II. BOT PROCESSOR DESCRIPTION

A. SCRIPT PROGRAMS AND LANGUAGE

30 As mentioned above, runtime executive 106 embodies the necessary information to maintain a reasonable conversation with human users to answer their inquiries and to carry on a general discourse with them on a set of topics. These topics are created by the back end in the form of script programs 108 that are compiled (or interpreted) and incorporated into runtime executive 106. In the preferred embodiment of the present invention, script programs may be

written by human designers having little or no formal programming experience. It will be appreciated that script programs can also be written by automated learning programs or through partially automated script creation tools, and that the present invention should not be limited to human-written script programs.

5 Two exemplars of such script programs are given below in Table 1.

=====

TABLE 1 -- TWO EXAMPLE SCRIPT PROGRAMS

EXAMPLE 1 --

10

```

Topic "CatsOrComputers" is
  IfHeard "cat","computer" then
    IfHeard "cat" then
      Say "What would you like to know about my cat?";
15      Focus "Cats";
      Done
    IfHeard "computer" then
      Say "What would you like to know about my computer?";
      Focus "Computers";
20      Done
    Done
  EndTopic

```

20

25

```

Topic "Cats" is
  IfHeard "cat","it" then
    IfHeard "mouse" then
      Say "It caught a mouse once and brought it",
        "to me as a present";
      Done
30      Done
    EndTopic

```

30

35

```

Topic "Computers" is
  IfHeard "computer","it" then
    IfHeard "mouse" then
      Say "The mouse is a PS/2 type mouse with three buttons";
      SwitchTo "Mouse Sales";
      Done
40      EndTopic

```

40

45

```

Topic "Unknown Input" is
  If ?WhatUserSaid DoesNotContain "cat","computer","mouse","it" then
    Say "Would you like to know about Cats or Computers?";
    Done
  EndTopic

```

```

Sequence Topic "Mouse Sales" is
  Always

```

Say "Would you like to buy one?";
 WaitForResponse;
 IfHeard "no" Then
 Done
 5 Say "OK, what is your name?";
 WaitForResponse;
 Remember ?UserName is ?WhatUserSaid;
 Say "That will be \$199.95",
 "Please enter your credit card number now";
 10 WaitForResponse;
 Remember ?CardNum is ?WhatUserSaid;
 Say "OK, We'll mail one to you within two weeks";
 "Please give me your mailing address now.";
 WaitForResponse;
 15 Remember ?Address is ?WhatUserSaid;
 Done
 EndTopic

Priority Topic "Swearing Filter" is
 20 IfHeard "fudge" Then // a popular swear word...
 Say "I'm sorry, that kind of language is not permitted here";
 Do "kick user off system";
 Done
 EndTopic

25

EXAMPLE 2 --

30

Topic "Price of XV17" is
 Subjects "XV17", "cost";
 IfHeard "XV17", "\$it" Then
 IfHeard "cost", "how much", "what about" Then
 35 Say "The XV17 monitor is now available for \$699";
 Done
 Continue
 EndTopic

40

Topic "Size of XV17" is
 Subjects "XV17", "features";
 IfHeard "XV17", "\$it" Then
 IfHeard "size", "how big", "what about" Then
 Say "The XV17 monitor has a 17 inch full-color screen";
 45 Done
 Continue
 EndTopic

50

Topic "Maker of XV17" is
 Subjects "XV17", "maker";
 IfHeard "XV17", "\$it" Then

IfHeard "who makes", "what about" Then
 Say "The XV17 monitor is made by NEC";
 Done
 Continue
 5 EndTopic

 Topic "Price of 5SG" is
 Subjects "5SG", "cost";
 IfHeard "5SG", "\$it" Then
 10 IfHeard "cost", "how much", "what about" Then
 Say "The 5SG monitor is now available for \$499";
 Done
 Continue
 15 EndTopic

 Topic "Size of 5SG" is
 Subjects "5SG", "features";
 IfHeard "5SG", "\$it" Then
 IfHeard "size", "how big", "what about" Then
 20 Say "The 5SG monitor has a 14 inch grayscale screen";
 Done
 Continue
 EndTopic

 25 Topic "Maker of 5SG" is
 Subjects "5SG", "maker";
 IfHeard "5SG", "\$it" Then
 IfHeard "who makes", "what about" Then
 Say "The 5SG monitor is made by MonitorTech";
 30 Done
 Continue
 EndTopic

 Topic "Price of 6SC" is
 35 Subjects "6SC", "cost";
 IfHeard "6SC", "\$it" Then
 IfHeard "cost", "how much", "what about" Then
 Say "The 6SC monitor is now available for \$899";
 Done
 40 Continue
 EndTopic

 Topic "Size of 6SC" is
 Subjects "6SC", "features";
 45 IfHeard "6SC", "\$it" Then
 IfHeard "size", "how big", "what about" Then
 Say "The 6SC monitor has a 19 inch full-color screen";
 Done
 Continue
 50 EndTopic

```

Topic "Maker of 6SC" is
Subjects "6SC", "maker";
  IfHeard "6SC", "$it" Then
    IfHeard "who makes", "what about" Then
      Say "The 6SC monitor is made by MonitorTech";
    Done
  Continue
EndTopic

```

Considering the script programs above, several aspects of the scripting language become apparent. First, as designed, the script language uses language primitives that are very close to human natural language. Thus, this scripting language is easy to use by users that have no facility in programming languages per se. TABLE 2 is a BNF (Backus Normal Form) description of the present embodiment of the scripting language:

TABLE 2 -- BNF DESCRIPTION OF THE SCRIPTING LANGUAGE WITH COMMENTS

```

<Program> = <Statement>*
<Statement> = <Definition> | <Category>

```

There are two types of statements in a program: constant definitions and input processing categories. All run-time processing of a user input is handled in the categories.

```

<Definition> = <PatternDef> | <PatternListDef> | <CategoryListDef> |
               <AttributeDef> | <OtherExampleDef> | <SubjectInfoDef>
<PatternDef> = Pattern <symbol> is <string>;
<PatternListDef> = PatternList <symbol> is <string> [, <string>*];
<CategoryListDef> = <CatListType> <symbol> is <string> [, <string>*];
<CatListType> = TopicList | ScenarioList | CategoryList;
<AttributeDef> = Attribute <memref>; | Attribute <memref> specificity
               <integer>;
<OtherExampleDef> = OtherExamples of <string> are <patlist>; |
                  OtherExamples of <string> WhenFocused are
                  <patlist>;
<SubjectInfoDef> = SubjectInfo <SubjectName> is
                  Replace <pronoun> with <replacement>
                  [, Replace <pronoun> with <replacement> |
                  , <pronoun> with <replacement>]*;
<SubjectName> = <string>
<pronoun> = <string>
<replacement> = <string>

```

Patterns are used to assign a name to a fixed string. The name can then be used in place of the string throughout the program, for readability and ease of modification. Similarly, a PatternList is used to assign a name to a list of strings, and a TopicList, ScenarioList, or

CategoryList is used to assign a name to a list of category names (see below.) Attribute declarations are used to declare attributes so that information about them can be displayed through various debugging functions. Declaration of attributes is optional; attributes can be used without being declared. An attribute declaration can also assign a “specificity” value that is used when the attribute is tested using **IfRecall** or any matching condition. If an attribute is not declared, or is not given a specificity value in its declaration, it is given the default specificity value of 2000. OtherExamples declarations define additional arguments for a particular example statement. These additional arguments are tested whenever the original example is tested using the automatic verification mechanism. An OtherExample declaration can also include the keyword WhenFocused to indicate that the arguments are context-sensitive examples.

A SubjectInfo declaration is used to assign pronoun replacement pairs to subjects, ordinarily subjects that have been assigned to one or more topics in the BOT script; the SubjectInfo declaration has no effect for subjects that are not defined. The pronoun and replacement can be any string. However, the invention is most commonly used for replacing the values of common English pronouns such as “he”. It is illegal for the same pronoun to be included more than once for a given subject, or to declare subject information for a subject more than once.

```

<Category> = <Topic> | <Scenario>
<Topic> = <CategoryInfo> Topic <string> is <Tstatement>* EndTopic
<Scenario> = <CategoryInfo> Scenario <string> is <Sstatement>*
              EndScenario
<CategoryInfo> = [Suppressed] [Priority | Default | Sequence]

```

A category is either a topic or a scenario. A topic is used to process user statements, while a scenario is used to process user actions. The term “category” is used to generically refer to a topic or scenario.

Categories are divided into four types, priority, standard, default, and sequence, according to the label preceding the word “topic” or “scenario”. A category that is not labeled is a Standard type. When the user makes a statement or takes an action, the categories in the program are executed, until a Done is reached (see below.) All priority categories are executed first, in the order in which they appear in the program. Next, all standard categories are executed. The order in which standard categories are executed changes dynamically depending on the execution of the program, and is described in the next paragraph. Finally, all default categories are executed, in the order in which they appear in the program. Sequence categories are executed only when explicitly accessed in a SwitchTo statement.

Standard categories are executed according to a “best-fit” matching mechanism, in which ties are broken according to an ordered list that initially corresponds to the order in which they appear in the program. When a standard category is executed, it, and other categories that share at least one Subject, is moved to the front of the standard category list (and so will be executed first on the next input.) The order of the standard category list can also be changed by commands within the program, as described below.

Categories can also be temporarily suppressed, in which case they are not executed at all. If the keyword Suppressed appears in front of the category definition, it is initially suppressed. Category suppression is discussed further below.

```

<Tstatement> = <MemoryLock> | <SubjectList> | <Tconditional>
<Sstatement> = <MemoryLock> | <SubjectList> | <Sconditional>
<Tconditional> = <Condition> (<Command> | <Tconditional>)*
                  <TconditionalEnd> |
                  <Tconditional> Otherwise <Tconditional>

```

**<Sconditional> = <Condition> (<Command> | <Sconditional>)*
 <SconditionalEnd> |
 <Sconditional> Otherwise <Sconditional>
 <TconditionalEnd> = Done | Continue | NextTopic | TryAgain | SwitchBack
 <SconditionalEnd> = Done | Continue | NextScenario | TryAgain | SwitchBack**

The body of each category is a list of conditional blocks. These conditional blocks are executed in the order found in the category. If the condition of a conditional block is false, execution goes on to the next conditional block in the category, or to the next category if there are no further conditional blocks. If the condition is true, the commands and conditional blocks inside the block are executed, and further behavior of the program is dependent on the keyword which ends the conditional block. If it ends with Done, execution ceases until the next input occurs (unless an InterruptSequence has been executed; see below.) If it ends with Continue, execution continues with the next conditional block in the category, or the next category if there are no further conditional blocks. If it ends with NextTopic/NextScenario, the rest of the current category is skipped and execution continues with the next category. If it ends with TryAgain, the most recent WaitForResponse within the block is executed (it is an error to end a block with TryAgain if it does not contain a WaitForResponse.) If it ends with SwitchBack, execution resumes immediately following whichever SwitchTo statement switched to the current block. It is an error to end a block with SwitchBack if the block is not inside a Sequence topic.

Conditional blocks can be combined using the Otherwise keyword; if the first condition is true then the condition block(s) that follow the Otherwise keyword are not executed. This behavior is similar to the behavior of an “else” command in C and similar programming languages.

<MemoryLock> = MemoryLock <memref> [,<memref>]*;

The top level of a category may contain one or more `MemoryLock` statements. Each `MemoryLock` statement asserts that the value of one or more associative memory elements should only be changed within that category. If an associative memory key `?x` is `MemoryLocked` in a category `C`, it is an error for a program to assign a value to `?x` using `Remember` or `Forget` anywhere outside the category `C`, or to `MemoryLock` `?x` in some other category.

<SubjectList> = Subjects <string> [,<string>]*;

The top level of a category may contain one or more Subjects statements. Each asserts that the given subjects are subjects of the topic. If a non-IF command within the body of the topic is executed, all topics which share at least one Subject with the topic are brought to the front of the focus of attention.

```

<Condition> = <SingleCondition> Then
| <SingleCondition> [and <SingleCondition>]* Then
| <SingleCondition> [or <SingleCondition>]* Then
| If <ConditionClause> [and <ConditionClause>]* Then
| If <ConditionClause> [or <ConditionClause>]* Then
| IfChance Then
| Always

```

A condition can either be a basic condition (described below) or a Boolean combination of basic conditions. A Boolean combination of basic conditions that includes both **and** and **or**

keywords must use parentheses to prevent possible ambiguity; there is no built-in operator precedence between **and** and **or** in GeRBiL. The Boolean **not** operator is implemented within the basic conditions; there is no explicit **not** keyword that applies to conditions. Finally, there are two basic conditions that cannot be combined using Boolean operators. The **IfChance** condition with no numeric argument is a probabilistic condition that has the same likelihood of being true as all the other argument-less IfChance statements immediately before or after it. Finally, the **Always** condition is simply always true.

```

<ConditionClause> = <MatchLHS> <PosMatchKeyword>
                    <MatchingList>
| <MatchLHS> <NegMatchKeyword> <PosMatchingList>
| Heard <MatchingList>
| NotHeard <PosMatchingList>
| Recall <MemList>
| DontRecall <PosMemList>
| Chance <chance>
| Focused
| (<ConditionClause> [and <ConditionClause>]*)
| (<ConditionClause> [or <ConditionClause>]*)
| {<ConditionClause> [and <ConditionClause>]*}
| {<ConditionClause> [or <ConditionClause>]*}

```

```

<MatchLHS> = <string> | <memref> | <starbufref>
<PosMatchKeyword> = Contains | Matches |
                    ExactlyMatches
<NegMatchKeyword> = DoesNotContain | DoesNotMatch |
                    DoesNotExactlyMatch

```

There are four basic types of condition clause. First, conditions using the match keywords match a particular input pattern, most normally an element of the user memory, such as the string said by the user, to some set of template patterns, which may contain various "macro" characters, such as wildcard characters. Negated matching keywords, such as **DoesNotContain**, are given their own special category, in order to prevent "double negative" conditions. The **Heard** and **NotHeard** keywords are shortcuts equivalent to the commonly used condition "?WhatUserMeant Contains". Second, **Recall** and **DontRecall** are used to test whether elements of the user memory have been set or not, and are most commonly used in practice for testing flags that are set by libraries, for instance to indicate the type of question or statement that is being processed. Third, **Chance** conditions are true or false at random with the given probability. Fourth, **Focused** conditions are true if the category in which it is found contains one or more subjects in common with a category that was activated by the most recent input that activated at least one subject. Condition clauses can also be combined using **and** and **or** as long as parentheses are used to prevent ambiguity. The curly bracket symbols {} can be used to indicate that a condition is optional.

```

<SingleCondition> = IfHeard <MatchingList> |
                    IfNotHeard <PosMatchingList> |
                    IfRecall <MemList> |
                    IfDontRecall <PosMemList> |
                    IfChance <chance>

```

The single condition objects are equivalent in meaning to the analogous condition objects, except that the If keyword is combined with the condition keyword. In the present implementation, there are also certain cases where single condition objects can be substituted for condition clause objects.

```

5      <MatchingList> = <MatchingListArg> [[and|&]
                        <MatchingListArg>]*
      | <MatchingListArg> [[and|&] <MatchingListArg>]*
        [[and|&] not <MatchingListArg>]*
10     | <MatchingListArg> [[or|,] <MatchingListArg>]*

      <MatchingListArg> = <patlistobj> | (<MatchingList>)
      <PosMatchingList> = <PosMatchingListArg> [[and|&]
                        <PosMatchingListArg>]*
15     | <PosMatchingListArg> [[and|&] PosMatchingListArg]*
        [[and|&] not <PosMatchingListArg>]*
        | <PosMatchingListArg> [[or|,] <PosMatchingListArg>]*

      <PosMatchingListArg> = <patlistobj> |
20     (<PosMatchingList>)

```

A matching list is a list of pattern list objects (single expressions evaluating to lists of strings; see below) separated by **and**, **and not**, or **or**. (The keyword **and** and the ampersand character (&) are interchangeable, as are the keyword **or** and the comma.) A matching list serves as the right-hand-side of a matching expression. Parentheses must be used to prevent ambiguity any time a memory reference list contains both **and** and **or**. Finally, a positive-only matching list does not allow the use of **and not**, in order to prevent double negatives such as "DoesNotContain X and not Y".

```

30     <MemList> = <MemListArg> [[and|&] <MemListArg>]*
                | <MemListArg> [[and|&] <MemListArg>]* [[and|&] not
MemListArg]*
                | <MemListArg> [[or|,] <MemListArg>]*
35

      <MemListArg> = <memref> | (<MemList>)
      <PosMemList> = <PosMemListArg> [[and|&]
                        <PosMemListArg>]*
                        | <PosMemListArg> [[or|,]
40     <PosMemListArg>]*

      <PosMemListArg> = <memref> | (<PosMemList>)

```

A memory reference list is a list of one or more memory references separated by **and**, **and not**, or **or**. (The keyword **and** and the ampersand character (&) are interchangeable, as are the keyword **or** and the comma.) Parentheses must be used to prevent ambiguity any time a memory reference list contains both **and** and **or**. Finally, a positive-only memory reference list does not allow the use of **and not**, in order to prevent double negatives such as "DoesNotContain ?X and not ?Y"

```

50     <Command> = Say <patlist>; | SayOneOf <patlist>; |

```

```

5      Do <patlist>; | DoOneOf <patlist>; |
      SayToConsole <patlist>; | SayToFile <string> <patlist>; |
      Trace <patlist>;
      Focus <catlist>; | Focus Subjects <string> [, <string>]*; |
10     DontFocus; | Suppress <catlist>; | Recover <catlist>; |
      Forget <memlist>; | ForgetOneOf <memlist>; |
      Remember <memlist>; | RememberOneOf <memlist>; |
      Remember <memref> is <patlist>; |
      Remember <memref> IsOneOf <patlist>; |
15     Remember <memref> is Compute <FunctionName> of
          <patlist>; |
      WaitForResponse; | InterruptSequence; |
      SwitchTo <string>; | SwitchTo <symbol>; |
      SwitchToOneOf <catlist>; |
20     Example <patlist>; | InitialExample
          <integer> <patlist>; |
      SequenceExample <exampleindex> <patlist>;

<FunctionName> = SpellCheck | URLEncoding |
20     ReplacePronouns |
          Capitalize | UpperCase | LowerCase

```

There are currently 27 basic commands. Say makes a statement to the user, while Do takes an action of some sort. (The possible arguments of a Do action are domain-specific.)

25 SayOneOf and DoOneOf nondeterministically select one of their arguments, and Say or Do that argument. SayToConsole is a Say statement whose output is directed to the console window and log file. SayToFile is a Say statement whose output is directed to a file specified in the command. Trace is a Say statement whose output is directed to the console window and log file, and only appears when the script is being run in various debugging modes. Remember is used to

30 assign values to associative memory elements; if a list of arguments is given with no is keyword, each argument is assigned an arbitrary non-empty value (currently the string "TRUE"). Remember can also be used to compute a function and assign its value to a memory element; currently implemented functions include spell-checking, URL encoding, pronoun replacement (according to pronoun-replacement pairs defined in SubjectInfo), and several string

35 capitalization operations. Forget is used to un-assign values of associative memory elements. Once Forget ?x has been executed for some element ?x, ?x will have no value and will not cause an IfRecall statement to become true, until a Remember statement is executed for ?x. ForgetOneOf, RememberOneOf, and Remember..IsOneOf are the nondeterministic equivalents of Forget, Remember, and Remember..Is, respectively. Suppress takes a list of categories as

40 arguments and suppresses each of its argument categories so that they are no longer executed on any input. Recover takes a list of categories as arguments and reverses the effect of a Suppress command. Focus takes a list of categories as arguments and places them at the front of the ordered category list. Focus Subjects takes a list of subjects as arguments and places all categories which cover at least one of those subjects (as defined with a Subjects command in the

45 top level of the category) at the front of the ordered category list. WaitForResponse halts execution in the same way as a Done statement but resumes at the same point on the next input. InterruptSequence can only be used within a Sequence topic, and temporarily halts execution of the current topic while all of the standard and default topics are executed. When a Done is reached, or when all of the standard and default topics are completed, execution resumes,

50 without waiting for further input, within the Sequence topic. A SwitchTo command immediately transfers control of execution to the named category. A SwitchToOneOf command chooses one

of its arguments at random and executes a SwitchTo on it. Example statements do not have any immediate effect, but are used in automatic verification.

<pat> = <string> | <symbol> | <memref> | <starbufref> | <pat> + <pat>

A pattern is anything that evaluates to a string. It can be an explicit string (indicated with quotes), the name of a Pattern object, an associative memory reference, a reference to a "star buffer element" (set according to wildcard characters appearing in template patterns within pattern matching conditional statements), or a concatenation of any of the above.

**<patlistobj> = <pat> | <symbol> | (<patlist>) | {<patlist>} |
<patlistobj> + <patlistobj>**

A patternlist object is any single expression that evaluates to a list of zero or more strings. It can be a single pattern, the name of a PatternList object, a PatternList enclosed in parentheses (also known as an "implicitly defined PatternList" since it is never explicitly given a name), a PatternList enclosed in curly brackets (indicating that the element or elements included within the brackets are "optional"), or a concatenation of any of the above. The value of the concatenation of two lists of strings is a list consisting of the concatenation of each element in the first list with each element of the second list. A symbol is a string of alphanumeric or underscore characters, beginning with a letter. Symbols are not case sensitive.

<patlist> = <patlistobj> [, <patlistobj>]*

A pattern list is anything that evaluates to a list of strings. It consists of one or more PatternList objects, separated by strings. Since each PatternList object may have a value that is a list of strings, the value of the PatternList is the value of all the elements appended together.

**<catlist> = <catname> [, <catname>]*
<catname> = <string> | This | <symbol>**

A category reference is either an explicit string containing the name of a category, the keyword This (referring to the category in which it appears) or the name of a CategoryList (or TopicList or ScenarioList) object. A category list is simply a list of categories or CategoryList objects separated by commas.

**<memref> = ?<symbol> | ?<pat>:<symbol>
<memlist> = <memref> [, <memref>]***

A reference to the associative memory is normally indicated by a ? followed by the name of the key. Such references are normally particular to the user whose input is being processed. A reference to the associative memory for another user can be made by putting a pattern referring to the other user between the ? and the key. The reference to the other user is separated from the key by a colon. A memory reference list is simply a list of memory references separated by commas.

<starbufref> = #<integer> | *<integer> | %<integer> | &<integer> | *match

The "star buffer" contains the substring of an input string which matched each *, #, %, or & wildcard character in the template pattern in the most recent successful match. References to this star buffer consist of a symbol (*, #, &, or %) followed by a number. *n refers to the

substring which matched the Nth * wildcard character found in the template, and so on. *match refers to the substring of the input string that matched the entire template pattern.

<chance> = <realnumber> | <realnumber>%

The argument of a Chance statement is either a real number between 0 and 1, interpreted as a probability, or a real number between 0 and 100 followed by a % sign, interpreted as a probability multiplied by 100.

<exampleindex> = <integer> [.<symbol>]*

The index for a SequenceExample statement is an integer followed by zero or more strings of alphanumeric characters, separated by periods.

=====

The second aspect of the example script programs is that the scripts themselves embody a particular universe of discourse reflective of the subject matter concerning the site itself -- e.g. a BOT for a site of a reseller of personal computer should "know" something about computers and their peripherals. These script programs are written in an action-response type style wherein the actual language supplied by the user embodies an "action" to which the "response" is written into the script program itself.

Scripts in the present embodiment are written generally by site administrators (human or otherwise) by defining a list of "categories" in which the site will be well conversant. Categories may comprise "topics" that are recognizable by the runtime executive. Topics, in turn, may comprise patterns or words that are matched against the stream of input communication (in either spoken or written or any other suitable form of communication) from the user.

To embody this knowledge into the runtime executive itself, the script programs are compiled by compiler 107 in **Figure 1**. As previously mentioned, these script programs may be iteratively tweaked to improve the interaction with human users by a re-edit and re-compile process. It will be appreciated that compiler techniques sufficient to implement the above-listed BNF language description are well known to those skilled in the art and that the present invention should not be limited to any particular compiler techniques.

B. RUNTIME EXECUTIVE PROCESS

Figure 4 expands the view of runtime executive 106 of **Figure 1**. Runtime executive 106 comprises local variables 402, robot object 404, and a list of user records 406. Robot object 404 is that part of runtime executive 106 that is incorporated by the compilation process described above. Although robot object 404 may be changed via the re-edit and re-compilation process as

mentioned, during runtime, robot object 404 typically does not change whilst in conversation with user 116. The list of user records 406 is provided because the BOT processor could be in conversation with multiple users simultaneously and therefore needs to maintain the state of each on-going conversation. The state for each such conversation is maintained in a user record 406.

- 5 Similar user records can also be used to store the state of the robot and as test users for automated testing routines. Finally, runtime executive 106 maintains local variables 402 that are used during the processing of a single user input.

TABLE 3 is a listing of the C++ header file that embodies runtime executive 106.

=====

10 TABLE 3 -- C++ HEADER FILE OF RUNTIME EXECUTIVE

```

class CProgram
{
public:
15     CExeStruct*           ProgramExecutable;
        CMatcher*          Matcher;
        CBFMatcher*        BFMatcher;
        // holds all short-term run-time data
        CRunStruct*         RunTime;
20     CGRBLToolDoc*        OwnerSession;
        FILE*               m_pfRSP;
        CString             CurrentInputString;
        // Registered attributes
        CTypedPtrMap<CMapStringToPtr, CString, CAttributeInfo*>
25         *m_pmspAttributeRegistry;
        // Subject information
        CTypedPtrMap<CMapStringToPtr, CString, CSubjectInfo*>
         *m_pmspSubjectInfo;
        // Shortcut to a special user used for testing; this user is in
30         // the general user records list as well.
        CUserRec*           TestUser;
        // Run-time options
        BOOL                TraceOn;
        BOOL                FullTraceOn;
35         BOOL                EchoOn;
        CategoryExecutionMode ExecutionMode;
        // TestMode TRUE indicates that user state is saved in order to go back and
        // retrieve further categories that were activated by an execution. (This also
        // happens if ExecutionMode is EqualMatches or AllMatches.)
40         // Used for Example mode and other debugging modes.
        BOOL                TestMode;
        BOOL                BEST_FIT_DEBUG;
        // whether it says anything when example answer is right
        BOOL                CorrectExampleTrace;
45         // so we don't try to send out to clients.

```

```

        BOOL                                ReplayingLog;
        // in order to have output and report messages echoed to a report file,
        // set Reporting to TRUE and set m_strReportFileName.
        // will cause output to be printed to m_strReportFileName.
5         BOOL                                Reporting;
        CString                                m_strReportFileName;
        // Values for keeping statistics during Example testing
        int                                    nExamplesRun;
        // correct answer not given, maybe others given
10        int                                    nWrongAnswers;
        int                                    nExtraAnswers;

        // Run is bound to a particular session doc and view, and executes
        // either Topics or Scenarios. If TestMode is on, produces no
15        // direct output.
        void Run(        CGRBLToolDoc*        ThisDoc,
                        CGRBLToolView*        ThisView,
                        LPCTSTR                TextLine,
                        LPCTSTR                UserName,
20                        ULONG                UserID,
                        ULONG                ProtocolMessageType);

        // Runs all examples in a file or program.
        void RunAllExamples(        CGRBLToolDoc*        Context,
                        LPCTSTR                InputFileName,
                        LPCTSTR                ReportFileName,
25                        BOOL                bFindAllMatches,
                        BOOL                bEchoOn,
                        BOOL                bPrintAll,
                        BOOL                bTraceOn);

        void RunSequenceExample(CSequenceExample*        ThisExample,
                        CUserRec*                ExampleUser,
                        CGRBLToolDoc*        Context,
                        CGRBLToolView*        pSessionView);
30        void RunExampleSet(        CExample*                ThisExample,
                        CUserRec*                ExampleUser,
                        CGRBLToolDoc*        Context,
                        CGRBLToolView*        pSessionView);
35        void RunExampleInput(        LPCTSTR                ExampleText,
                        CExample*                ThisExample,
                        CUserRec*                ExampleUser,
                        CGRBLToolDoc*        Context,
                        CGRBLToolView*        pSessionView);
40        // Functions to summarize all the examples in the bot
        void SummarizeExamples(LPCTSTR FileName);
        void SummarizeExample(CExample* ThisExample, FILE* f);
45        // Runs an attribute check
        void RunAttributeCheck(LPCTSTR InputText);
        // Performs "intelligent find-in-files"
        void LookForCategories(LPCTSTR InputList, int NumCategories, int Flags);
50        CSearchPattern* ConvertPatternToSearchPattern(CString Pattern, BOOL
        Subword);

```

```

// function which fetches the next category to be executed
CCategory* GetNextCategory( CGRBLToolDoc* Context,
                           CUserRec* ThisUser,
                           CCatType ExecutionType,
5                           CABlockEnd LastReturnVal);
// Output interface between the Bot and the Connector. This now
// outputs only to the robot, not to the console. The output message,
// annotated with "Robot says", etc., are put in RunTime->RobotOutputSummary
void RobotOutput(LPCTSTR TextLine,
10                  ULONG ThisUserID,
                  ULONG MsgType);
// Wrapper which uses the RunTime SayBuffer if needed
void BufferedSay(LPCTSTR TextLine,
15                  ULONG ThisUserID,
                  ULONG MsgType,
                  BOOL IsBuffered,
                  CArgListElem* ItemSaid);
// produces appropriate trace messages for example mode.
void HandleOutputInExampleMode(CAction* Action,
20                  CArgListElem* OutputItem);
// Output a line to the console. ALL output or potential output to
// the console and/or log or report files should go through this function.
void ConsoleOutput(OutputLineType MessageType,
25                  LPCTSTR Message,
                  LPCTSTR SourceFile,
                  int SourceLine);
void ConsoleOutput(OutputLineType MessageType,
                  LPCTSTR Message);
void ConsoleOutput(OutputLineType MessageType,
30                  CConsoleData* MessageData);
// pushes output to the console
void PushConsoleOutput(ConsoleOutputType OutputType);
void ClearConsoleOutput();
void PushConsoleOutputToString(ConsoleOutputType OutputType,
35                  CString& OutputString);
// version which bypasses all buffers and just prints it out.
void DirectConsoleOutput(OutputLineType MessageType,
                          LPCTSTR Message,
                          LPCTSTR SourceFile,
40                          int SorceLine);
void DirectConsoleOutput(OutputLineType MessageType,
                          LPCTSTR Message);
void DirectConsoleOutput(OutputLineType MessageType,
                          CConsoleData* MessageData);
45
// Reset user to initial state (of memory and attention stack)
void RestartUser(CUserRec* ThisUser);
// Special functions are declared here...
void DumpMemory(ULONG ID);
50 void PrintCurrentFocus(CUserRec* User, BOOL ShortPrint);
// Prime the random number generator for this thread

```

```

void PrimeTheRNG();
// Handle the refocusing component of the program execution
void Refocus();
// Continuation help functions
5 void SetupContinuation(CGRBLToolDoc* Context, CUserRec* ThisUser,
CContinuation* ThisContinuation);
// Functions to Remember and Forget automatically-defined
// attributes for the current user.
void SetUserAttribute(LPCTSTR Key, LPCTSTR Value);
10 void UnsetUserAttribute(LPCTSTR Key);
// Automatic pronoun replacement
BOOL ReplacePronouns(CString OriginalText, CString& FinalText);
// Intelligent Tracing Functions
void AddConditionTraceData(LPCTSTR Message,
15 LPCTSTR SrcFileName, int SrcLine);
void EnterIfFrame();
void EnterSwitchFrame(LPCTSTR Message, LPCTSTR SrcFileName,
int SrcLine);
void ExitIfFrame();
20 void ExitSwitchFrame(LPCTSTR Message, LPCTSTR SrcFileName,
int SrcLine);
void ExitAllFrames();
void AddTraceMsg(LPCTSTR Message, LPCTSTR SrcFileName,
int SrcLine, BOOL FullTraceOnly);
25 void ActivateTrace(); // equivalent to a null trace message
void ActivateExampleTrace(); // version for Examples mode.
void ReplayTrace(BOOL FullTrace);
int GetSize();
void PrintSize();
30
CProgram(CGRBLToolDoc* pgttd);
~CProgram();

};
35 =====

```

In the code given in Table 3, robot object 404 corresponds to ProgramExecutable, which is of type CExeStruct. **Figure 5** expands the view of robot object 404 as shown in **Figure 4**. Robot object 404 comprises several types of categories. These categories inherently maintain a priority by which runtime executive 106 processes inputs. For example, in **Figure 5**, four types of categories are depicted: priority categories 510, standard categories 520, default categories 530, and sequence categories 540. When an input comes into the BOT processor, the input is processed through a series of categories. First, the priority categories are processed to determine whether there is a response that will be generated by the current input. These priority categories are processed, in the present embodiment, in the order in which they appear in the runtime

executive. This order is currently selected in turn by the actual order in which PRIORITY TOPICS are found in the script program. This processing continues through the standard and default categories. Standard categories are executed according to the mechanisms disclosed below. Default categories are executed in the actual order in which DEFAULT TOPICS are found in the script program. Sequence categories 540 are also included in the robot object 404 but are not executed unless explicitly executed by a SWITCH-TO statement as described below. In the present embodiment, sequence categories are typically employed to perform a number of pre-defined sequential communications with a user to effect a desired result. For example, having the BOT take an order for tickets to an event, how many such tickets, credit card information to purchase such tickets, etc. is readily implemented as a sequence category. Such a sequence category would be SWITCHed-TO if prompted by a user inquiry to buy tickets. It will be appreciated that other hierarchies of categories may be defined and order of execution selected. It suffices for the purposes of the present invention that some hierarchy of categories is defined and that one or more of the mechanisms disclosed below be employed using one or more of such categories.

Figure 5 also contains subject-name to category map 550, which describes the categories associated with each subject found in a SUBJECTS command in one or more categories. This map helps to implement the Focus Subjects command and automatic focus mechanism, as described below.

Figure 6 describes the content of a category 502. Category 502 comprises body 610 and subject names 630. Body 610 is a list of pointers to condition-action blocks. Such a condition-action block is a representation of an IF-THEN block found a script program. Subject names 630 are a representation of a listing of SUBJECTS that may optionally be found in a script program. As will be discussed in greater detail below, subject names 630 are used to focus the attention of the BOT processor on other categories similar to the category being processed.

Figure 7 expands the view of user record 406 as shown in **Figure 4**. User record 406 comprises category attention focus list 710, category suppress list 720, user attribute memory 730, continuation records 740, and active subject list 750. In the current embodiment of the present invention, attention focus list 710 is an ordered list comprising the standard categories 520 found in robot object 404. More generally speaking, however, an attention focus list could be implemented as a much broader list of any number of categories, as opposed to any single type of category. Indeed, for the purposes of the present invention, an attention focus list is an ordering of categories that, by virtue of their ordering, may affect the execution of an automatic interface program (i.e. BOT). It will be appreciated that all the "lists" and other structures

mentioned herein could be implemented in a wide variety of well known data structuring techniques. For example, in the present embodiment, lists are implemented as CTypedPtrLists, however, lists can be readily implemented in hash tables, arrays, linked lists, or other known methods. Thus, the scope of the present invention should not be limited to specific data structure and algorithm techniques and should include all well known design and implementation variants.

The ordering of categories within the attention focus list 710 may be different for different users and reflects the state of the BOT processor's conversation with the particular user. The categories at the top of the list 710 represent areas of discourse in which the BOT processor is currently focused. In the present embodiment, when a new user begins communications with the BOT processor, the attention focus list 710 for that new user corresponds exactly to the standard categories list 520 -- which in turn corresponds to the order in which TOPICS are found in the script program. As conversation between the user and the BOT processor continues, this ordering of categories in attention focus list 710 is reordered according to the topics discussed by the user.

Category suppress list 720 is a list of categories that have been suppressed explicitly in the script program. Suppression of categories can occur a number of ways: suppressed categories may be initially listed as suppressed in the script program or categories may be subsequently suppressed by execution of a particular action in a script program. If the user touches upon a suppressed topic, then the suppressed category is not executed by the BOT processor. This suppress feature allows the BOT creator to have greater control over the BOT's "personality" as presented to the user.

User attribute memory 730 allows the BOT processor to remember certain attributes of the user that it has learned during the course of the conversation. For example, the gender, the telephone number, the credit card number, the address of the user may be particular fields found in user attribute memory 730.

Continuation records 740 are used primarily when the BOT processor has interrupted the execution of a category and may eventually wish to resume execution of said category. Such interruptions can occur as a result of a WaitForResponse statement (in which case the BOT processor has made a query of the user and is awaiting a response), an InterruptSequence statement (in which case the BOT processor has temporarily halted processing of the current category), or a SwitchTo statement (in which case the BOT processor may eventually return to the category containing the SwitchTo statement after executing a SwitchBack command.) At such a point, continuation record 740 maintains the location of the execution of the script in

memory. Once the interruption is complete, execution continues at such location. It will be appreciated that there are other times in which it is desired to store such execution state.

List of active subjects 750 is a list of strings containing the names of all the subjects that are currently the “focus of conversation”. This set of subjects is the set of all subjects activated during the most recent bot response that activated one or more subjects, either through executing a category associated with one or more subjects (using the SUBJECTS keyword), or executing a FOCUS or FOCUS SUBJECTS command. Further details on the use of the active subjects list for resolving the context of an input are disclosed below.

Figure 8 expands the view of local variables 402 as found in **Figure 4**. Local variables 402 comprise active user record 810, active continuation record 820, category focus list 830, and activated subjects list 840. Active user record 810 is the user record 406 that corresponds to the user that is currently talking to the BOT processor. Active continuation record 820 is one of the continuation records 740, if any, that is copied over for the current execution. Category focus list 830 provides an intermediate store of recently activated categories and other categories associated with them. Categories are associated if they share at least one subject name as listed in 630 in **Figure 6**. Activated subjects list 840 provides an intermediate store of subjects associated with recently activated categories, or categories focused using a FOCUS or FOCUS SUBJECTS command.

III. EXECUTION OF GERBIL PROGRAMS

A. THE INTERNAL STRUCTURE OF A GERBIL PROGRAM

Now a more detailed explanation of both the structure and the execution of Gerbil programs in the present embodiment will be given. The session document (CGRBLToolDoc) contains all information about a BOT as it is being executed, including two member variables particularly relevant to the current disclosure:

```
CUserRecordSet*   m_pursUsers;
CProgram*         m_ppProgram;
```

The m_pursUsers structure contains information about each user that is presently conversing with the BOT (stored as a map from UserIDs to CUserRec structures), such as any facts remembered about that user and the focus of attention for that conversation. The m_ppProgram structure contains all the information about the BOT script itself, including the robot object created by the compilation process described above and various run-time variables needed during script execution. The session document also includes a number of other members

such as pointers to various display windows and dialogs used during execution, lists of file names used in the BOT script, and so on.

There are two relevant member variables of the present embodiment of a Gerbil program (CProgram):

```

5      CExeStruct*      ExeProg;
      CRunStruct*      RunTime;

```

The ExeProg contains an executable version of the Gerbil script. The RunTime structure contains variables that are used when executing the Gerbil script.

10 The CExeStruct contains the following relevant member variables:

```

      CCategoryList      PriorityCategories;
      CCategoryList      DefaultCategories;
      CCategoryList      SequenceCategories;
      CCategoryList      StandardCategories;
15     CMapStringToPtr     m_pmspSubjectMap;

```

Each CCategoryList contains a list of CCategory objects. Each CCategory contains a set of CConditionActionBlock objects, each with a condition and a list of CAction objects. A CConditionActionBlock is a type of CAction, so CConditionActionBlock objects can recursively
20 contain other CConditionActionBlock objects. A CCategory also contains a list of all the subjects discussed by the category.

The lists PriorityCategories, DefaultCategories, and SequenceCategories are fixed in ordering and are shared among all users. Each user record contains a copy of the list StandardCategories (see below) in which the ordering of categories can dynamically change
25 (according to the focus mechanism). The copy of StandardCategories in the CExeStruct is fixed in order and is used to create the initial copy of StandardCategories for each new user.

Finally, the CExeStruct contains a map m_pmspSubjectMap from each subject name to the list of categories that discuss that subject

In the present embodiment, the CRunStruct contains three relevant member variables:

```

30     CUserRec*      User;
      CTypedPtrList<CObList, CCategory*> FocusList;
      CStringArray   m_saActiveSubjects;

```

It also contains a number of temporary pointer variables, including Continuation,
35 ActiveCatPos, and SwitchToCategory, which are used in execution as described below. User is a pointer to the user record for the user involved in the current conversation. FocusList is used to store the list of categories that have been activated by the focus of attention mechanism during the current execution of the Gerbil script. It will be used at the end of the run to modify the

focus of attention for the current user, as described below. `m_saActiveSubjects` is used to store the list of subjects that have been activated by the focus of attention mechanism during the current execution of the Gerbil script. It will be used at the end of the run to modify the active subjects map for the current user, as described below.

5 The `CUserRec` contains information about the current user and the robot's conversation with the user. In particular, it contains a `CMapStringToPtr` containing the contents of the memory for the user, in which each attribute name is mapped to a list of strings representing the value of that attribute, and seven member variables relevant to the present mechanisms:

10	<code>CCategoryList</code> <code>CTypedPtrList<CObList, CCategory*></code> <code>CContinuation*</code> <code>CTypedPtrList<CObList, CContinuation*></code> <code>CTypedPtrList<CObList, CContinuation*></code> <code>CMapStringToString</code> <code>CMapStringToPtr</code>	<code>AttentionFocus;</code> <code>SuppressList;</code> <code>Continuation;</code> <code>SwitchContinuations;</code> <code>SequenceContinuations;</code> <code>m_mssReplacements;</code> <code>m_mspActiveSubjects;</code>
15		

AttentionFocus is a copy of the StandardCategories list from the program executable that describes the attention focus for the BOT's conversation with the current user. The order of the categories in this list may be different than the order in StandardCategories, due to the functioning of the focus mechanism. SuppressList is a list of pointers to the categories that are suppressed in the robot's conversation with the current user. SuppressList may include categories from the PriorityCategories, DefaultCategories, and StandardCategories list. `m_mssReplacements` is a mapping from certain words to other words, used in implementation of the pronoun replacement mechanism. `m_mspActiveSubjects` is a mapping from subject names to null pointers, indicating the set of active subjects associated with the current conversation. This mapping is used in implementing the **Focused** command disclosed below. Next, Continuation is NULL unless there is a WaitForResponse command that is currently active. In this case, Continuation points to a CContinuation structure that describes where in the script the WaitForResponse is located and how to resume execution from that point. Finally, the user record contains stacks of continuations that handle interruptions of a Sequence category and switches back from Sequence categories. SwitchContinuations contains a CContinuation for each SwitchTo statement for which a SwitchBack is still possible (much like the call stack in other programming languages), while SequenceContinuations contains a CContinuation for each sequence that has been interrupted by an InterruptSequence command and not yet returned. The functioning of these CContinuation stacks is described further below.

B. THE EXECUTION OF A GERBIL PROGRAM

One main feature of a Gerbil program is its ability to “focus” categories for the express purpose of being more responsive to user communication. The “focusing” of categories, for the purposes of the present invention, is implemented by a combination of explicit and automatic methods. Explicit focusing can be accomplished in one of two ways in the current embodiment. The first focus mechanism, the “Focus” command, is added to the script program to explicitly focus a particular category when the command is executed. As will be explained below, “focusing” in the current embodiment moves the focused category to the front of the attention focus list. Thus, during the course of execution, the runtime executive will generally check the newly focused category earlier than it would have had the category not been focused. As an example, a sample Focus command might look like -- Focus “dogs”, “cats”; -- this command would move the category “dogs” to the front of the attention focus list and the category “cats” immediately following it. The Focus command is useful to make certain categories more immediate in the course of conversation and, in particular as to the above example, if the user had recently spoken of “pets”.

The second explicit focus mechanism, the “Focus Subjects” command, is similar to the “Focus” command but differs in that it will move a set of unspecified categories, each said category sharing a Subject whereby the Subject is explicitly listed within a “Subjects” command within the category. For example, in a script that discussed the subject of pets, the command **Focus Subjects “dogs”** could be placed in any category and if said command is executed, then all categories explicitly listing “dogs” as a SUBJECT will be placed to the front of the attention focus list. This command is useful to focus related categories without having to explicitly list them all.

In addition to these explicit focus mechanisms, there is an automatic focus mechanism that works without use of explicit commands. If a category is activated by executing an output statement (such as **Say** or **Do**) within the category, then that category is moved to the front of the attention focus list. Additionally, in the current embodiment, if that category contains a Subjects command, then all other categories which share at least one of the arguments of the Subject command are also moved to the front of the attention focus list. It will be appreciated that other protocols could be observed upon automatic focusing of a category.

The focusing of categories is also used to determine a set of “active subjects” associated with the current state of the conversation. Any time an input is processed that leads to one or more categories being “focused”, either through explicit or automatic focus methods, and at least one of these categories is associated with one or more subjects, the set of active subjects is

cleared and replaced with the set of all subjects associated with any of the categories that were activated (including any subjects explicitly mentioned in a "focus subjects" command.) Further details on the use of the set of active subjects are disclosed below.

Another, somewhat related mechanism, "Suppress", is implemented in the current embodiment. "Suppress" is an explicit command that disables the activation of the categories named in the command for the remainder of the course of conversation with that user. Such categories can be placed back into consideration with the use of the "Recover" command. For example, the command – Suppress "dogs"; -- will suppress the category "dogs" from further discussion, even if an explicit Focus command would purport to move it to the front of the attention focus list.

Now a more detailed description of the current embodiment will be discussed. During execution, each Gerbil command in the present embodiment actually returns a CABlockEnd value that describes what the program should do following the command. This value is normally Continue, indicating that the program should continue by executing the next Gerbil command. It can also be one of the values Waiting, Done, NextCategory, Switch, SwitchBack, NotActivated, or RunTimeError. (The Done, Continue, and NextTopic "terminators" that appear at the end of a condition block in a Gerbil code are actually implemented as commands that do nothing other than return the appropriate CABlockEnd value.) In this context, the following is a discussion concerning six Gerbil commands that are relevant to the focus of attention mechanism: Focus, Focus Subjects, WaitForResponse, TryAgain, InterruptSequence, and SwitchTo.

Each Focus command in a Gerbil script has as arguments a list of categories. This list is converted by the compiler into a list of pointers to the actual categories. When the Focus command is executed, these pointers are copied to the end of the RunTime->FocusList structure (to later be brought to the front of the attention focus list.) In addition, all subjects associated with each category are added to the list of active subjects. The C++ code for CFocus::Execute is straightforward and is shown below.

```

POSITION pos = ArgValues.GetHeadPosition();
for (; pos != NULL; ) {
    ArgCategory = (ArgValues.GetAt(pos))->Category;
    ArgValues.GetNext(pos);
    if (ArgCategory != NULL) {
        TRACE("Putting Category \"%s\" on focus list\n",
            ArgCategory->Name);
        Context->m_ppProgram->RunTime->FocusList.AddTail(
            ArgCategory);
    }
    for (int k=0; k<ArgCategory->m_saSubjectNames.GetSize(); k++)

```

```

5      {
          Context->m_ppProgram->RunTime->
              m_slActivatedSubjects.AddTail(
                  ArgCategory->m_saSubjectNames[k]);
      }
      return Continue;

```

10 In order to execute a "Focus Subjects" command, the robot executive takes each argument and uses the map m_pmspSubjectMap found in the CExeStruct to determine which categories share that subject. Each of the categories contained in the m_pmspSubjectMap under the subject name is appended to the end of RunTime->FocusList. The name of the subject is also added to the RunTime->m_slActivatedSubjects list.

15 The WaitForResponse command causes execution on the current input to stop, but before that, sets up a CContinuation telling the Gerbil program where to restart when the next input is processed. This CContinuation is created by the compiler and stored in the CWaitForResponse statement. The code for CWaitForResponse::Execute is trivial; it simply copies the CContinuation pointer into RunTime->User->Continuation and returns Waiting.

20 A TryAgain command is simply a special case of WaitForResponse in which the CContinuation starts from the previous WaitForResponse rather than the TryAgain command. A TryAgain command is converted into an appropriate CWaitForResponse by the compiler.

25 An InterruptSequence command can only be used within a Sequence category, and causes the execution of the category to be suspended while all of the standard and default categories are executed. (InterruptSequence can only be used after a WaitForResponse, to prevent possible conflicts in which a category might be executed twice.) It is implemented by adding a CContinuation to the top of the SequenceContinuations stack (allowing nested interruptions within interruptions) and returning the value NextCategory.

30 Each SwitchTo command in a Gerbil script has the name of a single category as an argument. Again, this category name is converted into a pointer by the compiler. When the SwitchTo command is executed at run-time, this pointer is copied into a member variable RunTime->SwitchToCategory and the value Switch is returned. Furthermore, a CContinuation representing the SwitchTo is copied into User->SwitchContinuations so that the category can be resumed if the target category ends with a SwitchBack. The fact that User->SwitchContinuations is a stack allows arbitrarily deep series of SwitchTo and SwitchBack calls.

35 In order to prevent cycles in which a category in the attention list is repeatedly executed and then SwitchedTo from another category later in the attention list, the present embodiment of the program checks to make sure that the category has not already been executed before

returning any value. If it has already been executed, the value RunTimeError is returned instead. Such cycles can only occur with standard categories. The compiler will check all sequence categories and guarantee that cycles among them will not occur. This is done by viewing each category as a node in a graph and each SwitchTo as an arc, and doing depth-first search to detect cycles in the graph. A WaitForResponse before the SwitchTo eliminates the arc caused by that SwitchTo, as it will prevent cycles from occurring while processing a single input. The C++ code for CSwitchTo::Execute is shown below. The SwitchToOneOf command is a straightforward extension of SwitchTo.

```

10      CCategory* DestCategory = Destinations[selection]->Category;
      ASSERT(DestCategory != NULL);
      if ((DestCategory->Executed) && (DestCategory->Priority !=
SequencePriority))
      {
15          // run-time error to switch to an already-executed non-sequence category
          Context->m_ppProgram->PrintTraceMsg("ERROR", SrcFileName,
                                              SrcLine);

          return RunTimeError;
      }
      // record what category is being switched to in the run-time data structure
20      Context->m_ppProgram->RunTime->SwitchToCategory = DestCategory;
      // and remember where it was called from
      Context->m_ppProgram->RunTime->User->SwitchContinuations.AddHead(
          m_pccCallingLocation);
25      return Switch;

```

The next level of structure above single commands in a Gerbil script is a CConditionActionBlock. A CConditionActionBlock consists of a condition and a body consisting of a list of commands. When the CConditionActionBlock is executed, the condition is first evaluated. If it is false, the block returns NotActivated immediately. Otherwise, the body statements are executed in order (normally starting with the first statement, but starting with a later statement if the block is part of an active Continuation) until one returns a CABlockEnd value other than Continue. When some other CABlockEnd value is returned, it is passed on as the return value of the CConditionActionBlock.

A CCategory contains an ordered list of CConditionActionBlock objects, and is executed by executing the blocks in succession (normally starting with the first block, but starting with a later block if the CCategory is part of an active Continuation.) If a block returns the value NextCategory, Switch, SwitchBack, Waiting, Done, or RunTimeError, execution of the CCategory stops and the return value is passed on. If a block returns NotActivated, the next block is executed. If a block returns Continue, the next block is activated unless it is an Otherwise block or unless both the current and next blocks are IfChance blocks, in which case it

and all other IfChance blocks immediately following it are skipped. If the last block in the category returns Continue or NotActivated, execution of the category is complete and the value NextCategory is returned. Meanwhile, if the category is a standard category, any output command (currently all variants of "Say" or "Do") will cause a flag to be set in the category. If this flag is set at the end of CCategory::Run, the category is appended to the end of RunTime->FocusList so that it will be automatically moved to the front of the focus of attention list. Furthermore, the subjects associated with the category are appended to the list RunTime->m_saActivatedSubjects. This automatic focus allows the attention focus mechanism to function even without the use of Focus statements. It will be appreciated that other implementations might decide whether a topic should be automatically focused in a different way, for example by automatically focusing on any topic in which the condition in at least one CConditionActionBlock has value true, or any topic in which any action is executed.

This behavior can be overridden by including the command DontFocus in any of the blocks that should not trigger the automatic focus mechanism. Furthermore, if the category is given a list of SUBJECTS in the Gerbil script, when the category is focused using automatic focus, all other categories that share at least one SUBJECT with said category are also appended to the end of RunTime->FocusList and will be automatically moved to the front of the focus of attention list.

When a user enters an input, the function CProgram::Run is called. This function does a number of low-level tasks (such as setting RunTime->User) and then executes the Gerbil program. First, it clears FocusList and m_slActivatedSubjects so that it can keep track of categories that are focused on during the execution. To prevent possible ambiguities in the ordering of category executions, Focusing actions do not have any effect until the script is finished executing on the current input. It will be appreciated that other implementations of an attention focus mechanism might dynamically reorder the attention focus list during the processing of an input.

The CProgram is executed by repeatedly selecting and executing categories, as shown in the code fragment below from CProgram::Run. RunTime->ActivePriority and RunTime->ActiveCatPos are used to keep track of what category is currently being executed. Once execution is complete, RunTime->FocusList is used to move those categories that were activated or focused on during execution to the front of the Focus of Attention, focusing the robot's attention on these categories. The function CProgram::Refocus itself is straightforward, simply going through RunTime->FocusList, and for each element, removing it from its previous position in the attention focus list and placing it at the front of the list. If the list RunTime-

>m_saActivatedSubjects is non-empty, CProgram::Refocus also updates the active subject map for the current user, clearing the map and replacing it with the contents of RunTime-

>m_saActivatedSubjects.

```

5      // mark all categories as un-executed
      ThisUser->AttentionFocus.MarkUndone();
      ProgramExecutable->PriorityCategories.MarkUndone();
      ProgramExecutable->DefaultCategories.MarkUndone();
      ProgramExecutable->SequenceCategories.MarkUndone();

10     // Clean up focus list and do a bunch of other initialization tasks
      RunTime->InitializeForRun();
      // Execute all of the categories, in order.

      CABlockEnd ReturnVal = NextCategory;
15     CCategory* ActiveCategory = GetNextCategory(ThisDoc, ThisUser,
                                                ExecutionType, ReturnVal);
      while (ActiveCategory != NULL) {
          ReturnVal = ActiveCategory->Run(ThisDoc);
          ActiveCategory = GetNextCategory(ThisDoc, ThisUser, ExecutionType,
20                                         ReturnVal);
      }
      // (other tasks done here such as handling output buffers)
      // handle all focusing actions
      Refocus();
25

```

Most of the work involved in deciding which categories to execute is done inside of CProgram::GetNextCategory. GetNextCategory uses RunTime->ActivePriority, RunTime->ActiveCatPos, and the ReturnVal from the previous category, and selects the next category to execute. If ReturnVal is NextCategory, the program will simply select the next category from the CategoryList for the current ActivePriority (Priority, Standard, or Default), according to the selection mechanism operative for that category and switching to the next priority level if necessary. (Recall that the Priority and Default categories are found in the CExeStruct, while the standard categories are found in RunTime->User->AttentionFocus. Sequence categories are never executed unless activated with a SwitchTo command, so the list ExeProg-

```

35 >SequenceCategories is never executed directly.) If there is an active CContinuation remaining
    from a previous execution (due to a WaitForResponse), it is activated immediately after the
    Priority categories. CContinuations are activated by returning the appropriate category and
    setting RunTime->Continuation, which will cause execution of the category to begin at the
    appropriate place within the category rather than the beginning.

```

```

40     If ReturnVal is Switch, the target category (from RunTime->SwitchToCategory) is
    selected. If ReturnVal is SwitchBack, the first CContinuation from SwitchContinuations is

```

removed from the stack and used to choose a category and set up a continuation, and set RunTime->ActiveCatPos when necessary. (Since SwitchBack can only be used within a Sequence category, there is guaranteed to be at least one continuation in SwitchContinuations. The process is equivalent to the method of returning from a subroutine in other programming languages.) If ReturnVal is Waiting, execution ceases since a WaitForResponse has been executed. Similarly, if ReturnVal is RunTimeError, execution ceases and the stack of SwitchContinuations and SequenceContinuations is cleared. (RunTimeError is presently returned only in the event of a SwitchTo cycle violation.) Finally, if ReturnVal is Done (recall that a category cannot return value NotActivated or Continue), execution stops unless there was an InterruptSequence that has not yet been resumed. Recall that InterruptSequence stops execution of a Sequence category while all of the Standard and Default categories are executed, and then resumes execution of the Sequence category. Therefore, if a Done is reached while there is at least one CContinuation in the SequenceContinuations stack, that Sequence category is resumed. In the case where there is no SequenceContinuation, the SwitchContinuations stack can also be cleared, as there is no possibility of returning from any SwitchTo statements once a Done (that is not ending an interruption) is executed.

IV. EXAMPLES OF AUTOMATIC ATTENTION FOCUSING

Having described the architecture of a present embodiment, the operation of the present invention will now be illustrated by describing the execution of the two example script programs, listed above in Table 1. For the purposes of illustration, assume that a site visitor engages in the following discussion with a site that has implemented script example 1:

25 Transcript of Interaction with Robot incorporating script Example 1:

Console Says: hello

Robot Says: Would you like to know about Cats or Computers?

30 Console Says: cats

Robot Says: What would you like to know about my cat?

Console Says: has it ever caught a mouse

Robot Says: It caught a mouse once and brought it

35 Robot Says: to me as a present

Console Says: tell me about computers

Robot Says: What would you like to know about my computer?

Console Says: does it have a mouse
Robot Says: The mouse is a PS/2 type mouse with three buttons
Robot Says: Would you like to buy one?

5 Console Says: Yes
Robot Says: OK, what is your name?

Console Says: Aloysius
Robot Says: That will be \$199.95
10 Robot Says: Please enter you credit card number now

Console Says: 3333 3333 3333 3333
Robot Says: OK, We'll mail one to you within two weeks
Robot Says: Please give me your mailing address now.

15 Console Says: Fudge. That's too late
Robot Says: I'm sorry, that kind of language is not permitted here
(user is kicked off system)

20

Execution of the Sample Gerbil Script Example 1:

The Gerbil Script given in Example 1 contains 6 categories: one priority topic, one sequence topic, and four standard topics. The priority topic, "Swearing Filter", is placed in
25 ExeProg->PriorityCategories; the sequence topic, "Mouse Sales", is placed in ExeProg->SequenceCategories, and the four other topics are placed in ExeProg->StandardCategories, in the order in which they appear in the script. Once the user begins talking to the robot, the categories in ExeProg->StandardCategories are copied to the AttentionFocus field of the CUserRec for the user. It is assumed here that the same user is talking to the robot throughout
30 this example.

In the remainder of this example, the "focus of attention" is referred to as the list of all of the topics, in the order in which they are executed. (Note that this should not be confused with the AttentionFocus member variable of a CUserRec.) The initial focus of attention for this user is as follows (the position of "Mouse Sales" is arbitrary since it is not
35 executed unless specifically activated.)

Topic	Type	Where It Is
Swearing Filter	Priority	ExeProg->PriorityCategories
CatsOrComputers	Standard	RunTime->User->AttentionFocus
Cats	Standard	RunTime->User->AttentionFocus
Computers	Standard	RunTime->User->AttentionFocus
Unknown Input	Standard	RunTime->User->AttentionFocus
(Mouse Sales)	(Sequence)	ExeProg->SequenceCategories

>Console Says: hello

>Robot Says: Would you like to know about Cats or Computers?

- 5 The program begins by executing the categories in ExeProg->PriorityCategories. This list contains only one topic, "Swearing Filter." The condition in the IfHeard statement in the topic does not hold, so the commands in the body are not executed, and the topic returns the CABlockEnd value NextCategory. Next, it begins executing the categories in RunTime->User->AttentionFocus. First, it executes the topic "CatsOrComputers". The input contains neither
- 10 "cat" nor "computer", so the If block is not executed and the topic returns NextCategory. Similarly, no condition blocks in the topics "Cats" or "Computers" have conditions that are true for this input, so NextCategory is returned for each. Finally, it executes the topic "Unknown Input". The condition If ?WhatUserSaid DoesNotContain "cat", "computer", "mouse", "it" holds, so the body of the block is executed and the robot says "Would you like to know about
- 15 cats or computers?" Since a statement in the category was executed, a pointer to the category is added to RunTime->FocusList. Finally, the block returns Done, and the category thus returns Done. There are no more topics to execute (if there were, they would not be executed now because ReturnVal is set to Done), and no active SequenceContinuations, so CProgram::Refocus is executed and the topic "Unknown Input" is moved to the front of RunTime->User-
- 20 >AttentionFocus and the run is complete.

The focus of attention is now as follows:

Topic	Type	Where It Is
Swearing Filter	Priority	ExeProg->PriorityCategories
Unknown Input	Standard	RunTime->User->AttentionFocus
CatsOrComputers	Standard	RunTime->User->AttentionFocus
Cats	Standard	RunTime->User->AttentionFocus
Computers	Standard	RunTime->User->AttentionFocus
(Mouse Sales)	(Sequence)	ExeProg->SequenceCategories

> Console Says: cats

> Robot Says: What would you like to know about my cat?

5 Again, the priority topic "Swearing Filter" is executed and has no effect. Next, the topic "Unknown Input" is executed. The DoesNotContain condition does not hold, so the topic has no effect and returns NextCategory. Now, the topic "CatsOrComputers" is executed. The first condition

IfHeard "cat", "computer"

10 is true, as is the condition on the first block inside of it,

IfHeard "cat"

 Thus the robot says "What would you like to know about my cat?" and executes the command Focus "Cats". When it executes this Focus command, a pointer to the topic "Cats" is added to RunTime->FocusList. (Note that RunTime->FocusList was empty, since it is cleared at the beginning of each call to CProgram::Run.) Finally, the block returns Done. Since ReturnVal is now Done, the next block in the "CatsOrComputers" topic is not executed. Finally, since a statement within "CatsOrComputers" has been executed, the topic is auto-focused by adding a pointer to the topic to the end of RunTime->FocusList. Since ReturnVal is Done, no further categories are executed, and the program calls CProgram::Refocus, which moves the two topics "Cats" and "CatsOrComputers" to the front of the attention focus list for this user. The focus of attention is now as follows:

Topic	Type	Where It Is
Swearing Filter	Priority	ExeProg->PriorityCategories
Cats	Standard	RunTime->User->AttentionFocus
CatsOrComputers	Standard	RunTime->User->AttentionFocus
Unknown Input	Standard	RunTime->User->AttentionFocus
Computers	Standard	RunTime->User->AttentionFocus
(Mouse Sales)	(Sequence)	ExeProg->SequenceCategories

- > Console Says: has it ever caught a mouse
- > Robot Says: It caught a mouse once and brought it
- > Robot Says: to me as a present

5 Again, the priority topic “Swearing Filter” is executed and has no effect. The program now executes topic “Cats” as it is the first topic in AttentionFocus for this user. The outer IfHeard condition is true since the input contains both “it” and “mouse”. Thus, the robot says “It caught a mouse once and brought it to me as a present.” Since a command within “Cats” was executed, a pointer to “Cats” is put in RunTime->FocusList. The inner block returns Done, 10 so no further blocks or topics are executed. CProgram::Refocus is now executed, but since “Cats” is already at the front of AttentionFocus, the order of the topics in the focus of attention is unchanged.

- 15 > Console Says: tell me about computers
 > Robot Says: What would you like to know about my computer?

The topics “Swearing Filter” and “Cats” are both executed with no effect. The program now executes topic “CatsOrComputers”, and the if block with condition

IfHeard “computer”

20 is executed. The robot thus asks “What would you like to know about my computer?” and executes the Focus statement, which puts a pointer to “Computers” in RunTime->FocusList. Since the topic “CatsOrComputers” was activated, a pointer to it is also placed in RunTime->FocusList, after the pointer to “Computers”. Since the block ends with a Done, the value Done is returned and execution stops. The two topics “Computers” and “CatsOrComputers” are now 25 moved to the front of AttentionFocus for the user, and the focus of attention is now as follows:

Topic	Type	Where It Is
Swearing Filter	Priority	ExeProg->PriorityCategories
Computers	Standard	RunTime->User->AttentionFocus
CatsOrComputers	Standard	RunTime->User->AttentionFocus
Cats	Standard	RunTime->User->AttentionFocus
Unknown Inputs	Standard	RunTime->User->AttentionFocus
(Mouse Sales)	(Sequence)	ExeProg->SequenceCategories

- > Console Says: does it have a mouse
- > Robot Says: The mouse is a PS/2 type mouse with three buttons
- > Robot Says: Would you like to buy one?

5 After executing “Swearing Filter” without effect, it begins AttentionFocus, starting with the topic “Computers”. Both IfHeard conditions are true, since the input contains both “it” and “mouse”. Thus, the robot says “The mouse is a PS/2 type mouse with three buttons.” The SwitchTo statement is now executed. After checking that “Mouse Sales” has not already been executed, it sets RunTime->SwitchToCategory to point to “Mouse Sales” and

10 returns the CABlockEnd value Switch (it also adds a continuation to RunTime->User->SwitchContinuations, but since “Mouse Sales” does not contain a SwitchBack command, this continuation never gets used.) Thus, on the next iteration, GetNextCategory returns “Mouse Sales” (it would also have moved RunTime->ActiveCatPos if “Mouse Sales” had not been a Sequence topic.) The execution of “Mouse Sales” begins by saying “Would you like to buy

15 one?” and then executing the WaitForResponse command. The WaitForResponse command stores a CContinuation pointing to the following statement into RunTime->User->Continuation and returns the value Waiting, stopping execution. Meanwhile, the topic “Computers” was added to RunTime->FocusList but Refocus has no effect since “Computers” is already at the start of RunTime->User->AttentionFocus. The topic “Mouse Sales” is not a standard topic and

20 therefore is not ever added to RunTime->FocusList.

The focus of attention can now be viewed as follows. Note that “Mouse Sales” is placed immediately below “Swearing Filter” since, due to the active CContinuation, it will be executed before any of the topics in RunTime->User->AttentionFocus.

Topic	Type	Where It Is
Swearing Filter	Priority	ExeProg->PriorityCategories
Mouse Sales	Sequence	RunTime->User->Continuation
Computers	Standard	RunTime->User->AttentionFocus
CatsOrComputers	Standard	RunTime->User->AttentionFocus
Cats	Standard	RunTime->User->AttentionFocus
Unknown Inputs	Standard	RunTime->User->AttentionFocus

- > Console Says: Yes
- > Robot Says: OK, what is your name?

5 After executing "Swearing Filter" with no effect, the robot checks to see if there is an active continuation. Since there is, it sets RunTime->Continuation, and now checks to see if the continuation is from a sequence topic. Since it is, GetNextCategory selects "Mouse Sales" before any of the categories in RunTime->User->AttentionFocus. Execution resumes immediately following the first WaitForResponse (the category checks RunTime->Continuation to see where within its body to begin executing; if RunTime->Continuation were NULL, it would begin with the first block.) Thus, the robot checks that it didn't just hear "no", and says

10 "OK, what is your name?", then executes WaitForResponse, which sets up another CContinuation in RunTime->User->Continuation and returns Waiting, ending execution. In this execution, nothing was added to RunTime->FocusList so no refocusing is done. The focus of attention is identical to that shown after the previous input.

15

- > Console Says: Aloysius
- > Robot Says: That will be \$199.95
- > Robot Says: Please enter you credit card number now

20

- > Console Says: 3333 3333 3333 3333
- > Robot Says: OK, We'll mail one to you within two weeks
- > Robot Says: Please give me your mailing address now.

25

The processing of the next two inputs is identical to the processing of the "yes" earlier, and since the continuations remain within "Mouse Sales", the focus of attention remains the same.

30

- > Console Says: Fudge. That's too late
- > Robot Says: I'm sorry, that kind of language is not permitted here
- > (user is kicked off system)

As usual, the program begins by executing "Swearing Filter". Since the input contains the word "fudge", the condition in "Swearing Filter" is executed, and the robot says "I'm sorry, that kind of language is not permitted here" and executes the Do command that kicks the user off the system. Since this topic returns Done, CProgram::Run sets ReturnVal to Done, and the CContinuation of "Mouse Sales" is not copied into RunTime->Continuation and thus not executed on this run.

Transcript of Interaction with Robot incorporating script Example 2:

- 10 Console Says: How big is the 6SC?
Robot Says: The 6SC monitor has a 19 inch full-color screen
- Console Says: How much is it?
Robot Says: The 6SC monitor is now available for \$899
- 15 Console Says: What about the 5SG?
Robot Says: The 5SG monitor is now available for \$499
- Console Says: How big is its screen?
20 Robot Says: The 5SG monitor has a 14 inch grayscale screen
- Console Says: Who makes it?
Robot Says: The 5SG monitor is made by MonitorTech
- 25 Console Says: What about the XV17?
Robot Says: The XV17 monitor is made by NEC

The auto-focus mechanism that is used to automatically bring categories to the front of the robot's focus of attention can be supplemented by specifying one or more "subjects" for a topic. If subjects are specified for a category C, when C is executed, it will be brought to the front of the focus of attention, and all other categories that share at least one subject with category C will also be brought to the front of the focus of attention. This mechanism allows for the automatic execution of behavior that could otherwise only be generated by an explicit "Focus" statement in the Gerbil script listing all of the categories that cover similar subjects.

35 The subject-based auto-focus mechanism is useful for handling pronouns (as in the use of "it" in the sample transcript) and answering context-dependent questions (such as the questions "What about the 5SG?" and "What about the XV17?" in the sample transcript.)

Subject specifications are included in categories as a CStringArray of subject names. The compiler produces a CMapStringToPtr that maps each subject name to a pointer to a CCategoryList that contains all the categories associated with that subject name. The compiler then uses this map to set up a list of CCategory pointers for each category C that describes all

categories that share at least one subject with C. When the category C is executed, all the pointers in this list are added to RunTime->FocusList and will be moved to the front of the focus of attention at the end of execution. The CExeStruct also contains a

CMapStringToPtr(m_psmSubjectMap) that maps subject names to categories, for use in

5 executing a "Focus Subjects" command.

Execution of the Sample Gerbil Script Example 2:

10 As with the first example, it is assumed that this interaction occurs between the robot and a single user. The details of the execution of each topic for each input are the same as in the first example, with the exceptions noted in the previous section. Since this script contains only standard topics, all of the topics are in:

RunTime->User->AttentionFocus.

15 Therefore, our diagrams of the robot's focus of attention for this example will only show the ordering of the topics within RunTime->User->AttentionFocus, not the location or type of the topics. For convenience, our diagrams also show the list of subjects associated with each topic. The initial ordering of the topics is exactly as found in the Gerbil script:

Topic	Subjects
Price of XV17	"XV17", "cost"
Size of XV17	"XV17", "features"
Maker of XV17	"XV17", "maker"
Price of 5SG	"5SG", "cost"
Size of 5SG	"5SG", "features"
Maker of 5SG	"5SG", "maker"
Price of 6SC	"6SC", "cost"
Size of 6SC	"6SC", "features"
Maker of 6SC	"6SC", "maker"

20

- > Console Says: How big is the 6SC?
- > Robot Says: The 6SC monitor has a 19 inch full-color screen

25 The input does not contain "XV17", "5SG", or a word beginning with "it" (the "\$it" pattern matches any word beginning with the letters "it", such as "it" or "it's".) Therefore, none of the first six topics are activated. In the seventh topic, "Price of 6SC", the outer IF block is activated because "6SC" was heard, but the inner IF block is not activated, so the topic returns

NextCategory. Since no statement was executed in the topic other than the IF conditions, the flag in the topic indicating activation is not set, and nothing is added to FocusList. Finally, in the eighth topic, "Size of 6SC", the conditions of both the outer and the inner IF block are true, so the robot says "The 6SC monitor has a 19 inch full-color screen". This inner clause returns Done, so execution is complete. However, this statement also sets the activation flag in the topic, so that when the topic is completed, the topic is added to the front of RunTime->FocusList. Since the topic also has two subjects associated with it ("6SC" and "features"), all other topics associated with these topics are also added to RunTime->FocusList. The two other "6SC" topics, "Price of 6SC" and "Maker of 6SC" are the added first, followed by the two other "features" topics "Size of XV17" and "Size of 5SG". (The order is consistent with the order in which the topics were originally defined.) Once execution is complete, the topics in RunTime->FocusList are moved to the front of RunTime->User->AttentionFocus, producing the focus of attention described below:

Topic	Subjects
Size of 6SC	"6SC", "features"
Price of 6SC	"6SC", "cost"
Maker of 6SC	"6SC", "maker"
Size of XV17	"XV17", "features"
Size of 5SG	"5SG", "features"
Price of XV17	"XV17", "cost"
Maker of XV17	"XV17", "maker"
Price of 5SG	"5SG", "cost"
Maker of 5SG	"5SG", "maker"

- 15 > Console Says: How much is it?
 > Robot Says: The 6SC monitor is now available for \$899

The topic "Size of 6SC" is executed first. The outer IF condition is true, as the input contains the word "it", but the inner IF statement is not activated. Since no non-IF statement was executed, the topic returns NextCategory and nothing is added to RunTime->FocusList. Next, the topic "Price of 6SC" is activated. The outer IF condition is true since the input contains "it", and the inner IF condition is true since the input contains "how much". Therefore, the robot "knows" that the question is referring to the 6SC monitor, despite the fact that the user's question does not mention the type of monitor. It thus says "The 6SC monitor is now available for \$899", and the topic returns Done. (Note that the topics "Price of XV17" and "Price of 5SG" would also be activated by the same input, but since they are lower on the attention stack, they are not executed at this time.) Since a statement within the topic was executed, the topic is added to RunTime->FocusList. Since it has two subjects "6SC" and "cost", the other two topics with

subject "6SC" and the other two topics with subject "cost" are also added to RunTime->FocusList. Execution of the script for this input is now finished, so the topics in FocusList are moved to the front of the focus of attention, producing the following focus:

Topic	Subjects
Price of 6SC	"6SC", "cost"
Size of 6SC	"6SC", "features"
Maker of 6SC	"6SC", "maker"
Price of XV17	"XV17", "cost"
Price of 5SG	"5SG", "cost"
Size of XV17	"XV17", "features"
Size of 5SG	"5SG", "features"
Maker of XV17	"XV17", "maker"
Maker of 5SG	"5SG", "maker"

- 5 > Console Says: What about the 5SG?
 > Robot Says: The 5SG monitor is now available for \$499

The first three topics in the focus of attention are not activated, as the input does not contain "6SC" or a word beginning with "it". Similarly, "Price of XV17" is not activated. The next topic executed is "Price of 5SG". The outer IF condition is true because the input contains "5SG", while the inner IF condition is true because the input contains "what about". Therefore, the robot "knows" that, given the previous conversation, the user is asking about the price of the monitor despite the fact that the user's question said nothing about price. It thus says "The 5SG monitor is now available for \$499" and returns the value Done. As was done in the other case, the topic "Price of 5SG" is added to RunTime->FocusList, as are the four other topics associated with the subjects "5SG" and "cost". These subjects are moved to the front of the focus of attention, producing the following:

Topic	Subjects
Price of 5SG	"5SG", "cost"
Size of 5SG	"5SG", "features"
Maker of 5SG	"5SG", "maker"
Price of XV17	"XV17", "cost"
Price of 6SC	"6SC", "cost"
Size of 6SC	"6SC", "features"
Maker of 6SC	"6SC", "maker"
Size of XV17	"XV17", "features"
Maker of XV17	"XV17", "maker"

- > Console Says: How big is its screen?

> Robot Says: The 5SG monitor has a 14 inch grayscale screen

The outer IF condition in the first topic "Price of 5SG" is true because the input contains the word "its", but the inner condition is not true, so the topic is not activated. In the next topic, "Size of 5SG", the outer IF condition is true because the input contains "its" while the inner IF condition is true because the input contains "how big". Thus, the robot knows that "its" refers to the 5SG monitor (recall that in a previous input, "it" was correctly interpreted as referring to the 6SC monitor instead.) Therefore, the robot answers the question correctly and moves the topic "Size of 5SG" to the front of the attention focus, followed by the other two topics with subject "5SG" and the other two topics with subject "features". The new focus of attention is as follows:

Topic	Subjects
Size of 5SG	"5SG", "features"
Price of 5SG	"5SG", "cost"
Maker of 5SG	"5SG", "maker"
Size of XV17	"XV17", "features"
Size of 6SC	"6SC", "features"
Price of XV17	"XV17", "cost"
Price of 6SC	"6SC", "cost"
Maker of 6SC	"6SC", "maker"
Maker of XV17	"XV17", "maker"

> Console Says: Who makes it?

> Robot Says: The 5SG monitor is made by MonitorTech

The next statement is processed similarly, as the combination of "it" and "who makes" activates the third topic "Maker of 5SG" and produces the correct answer. The topic "Maker of 5SG" is moved to the front of the focus of attention, again followed by the other two "5SG" topics and the other two "maker" topics. The new focus of attention is as follows:

Topic	Subjects
Maker of 5SG	"5SG", "maker"
Price of 5SG	"5SG", "cost"
Size of 5SG	"5SG", "features"
Maker of XV17	"XV17", "maker"
Maker of 6SC	"6SC", "maker"
Size of XV17	"XV17", "features"
Size of 6SC	"6SC", "features"
Price of XV17	"XV17", "cost"
Price of 6SC	"6SC", "cost"

- > Console Says: What about the XV17?
- > Robot Says: The XV17 monitor is made by NEC

None of the first three topics are activated, as the input does not contain either "5SG" or any word beginning with "it". In the next topic, "Maker of XV17", the outer condition is activated because the input contains "XV17", while the inner condition is activated because the input contains "what about". Therefore, the robot answers the question by giving the maker of the XV17. (Recall that the previous "What about" question was correctly interpreted as referring to the price of the monitor, while this is correctly interpreted as referring to the maker of the monitor.) Once again, the topic "Maker of XV17" is moved to the front of the attention list, followed by the four other topics that share at least one subject with the topic "Maker of XV17". The final focus of attention is as follows:

Topic	Subjects
Maker of XV17	"XV17", "maker"
Price of XV17	"XV17", "cost"
Size of XV17	"XV17", "features"
Maker of 5SG	"5SG", "maker"
Maker of 6SC	"6SC", "maker"
Price of 5SG	"5SG", "cost"
Size of 5SG	"5SG", "features"
Size of 6SC	"6SC", "features"
Price of 6SC	"6SC", "cost"

V. IMPLEMENTATION OF AUTOMATIC RESPONSE SELECTION

A. OVERVIEW

The mechanism for automatic selection of an appropriate response does not require the BOT author to change the BOT scripts in any way. The BOT author simply writes routines to respond to inputs in the natural way, and the mechanism handles resolution of situations in which

an input matches multiple conditions. The BOT author can also add “optional” elements to a condition that do not change whether the condition is true but may affect the automated response selection.

5 In one embodiment of the invention, Priority and Default categories allow the BOT author to implement initial filters and default handlers for the input, as well as any response generation that the BOT author wishes to have handled by conditions that are explicitly ordered. Sequence categories also function as described, being executed if and only if they are explicitly switched to by another executed category. The mechanism for automatic response selection functions on all other categories in the BOT script. It would be obvious to one skilled in the art
10 that the automatic response selection mechanism could also be applied to other types of categories, and that the scope of the present invention should not be limited to any particular method of partitioning the set of categories in the BOT script.

In response to an input, we consider a category “activated” if one or more base-level statements (i.e. statements other than IF conditionals) would be executed if the category were
15 executed. When the BOT script is compiled, the compiler builds a data structure that can be used to map an input and BOT state to the set of all categories that are activated by that input and BOT state. At run-time, this structure is used to generate a list of all activated categories and to assign a numerical measure of appropriateness (variously termed “specificity” throughout the description or, more generally, a “metric”) to each category. The category with the highest
20 appropriateness value is executed; in the current implementation, ties are broken according to some selection function, such as the aforementioned Focus mechanisms. If this category executes a Done statement, execution is complete, else the process is repeated, excluding any already-executed categories, and a new category is chosen. If at any step of this process (including the first step), no new categories are generated and a Done has not yet been executed,
25 execution switches to the Default categories and proceeds in the standard manner.

It will be appreciated that other implementations of the above mechanisms are possible. For example, the BOT could apply the numeric measure or metric to each category defined within the BOT without regard to activation of categories. As suggested above, the activation status of categories could be computed concurrently with the computation of the measure or
30 metric; or these steps could be performed separately. The present invention should not be limited to any one such implementation and the scope of the present invention explicitly includes these obvious implementation variants.

B. THE SPECIFICITY MEASURE FOR CONDITIONS

The present implementation of the invention computes the appropriateness of a particular response based on the estimated likelihood of the condition or conditions that suggested that response. Conceptually, the system computes the most specific condition that matched the input. For instance, the condition that looks for the pattern “you*sales bot” is more specific than the condition that looks for the pattern “you*bot”. The “*” is a wildcard symbol that can match zero or more words. This measure, known as “specificity”, is based on $\log(1/f)$ where f is the estimated likelihood, over all expected inputs to the system, that a condition is true for any particular input. In the present implementation, specificity is multiplied by 1000 to allow the computations to be done using integers. This inverse frequency measure correspond to the idea that a “more specific” question is composed of more and/or more unusual words than a “less specific” question. It will be appreciated that other measures of the specificity of a condition, including other formulas based on condition frequency or estimates derived directly from an explicit domain description, could be used and that the present invention should not be limited to the specificity measure described above.

The frequency of conditions, f , can be estimated in a variety of ways; for instance, words can be considered uniformly likely, a preset dictionary of word frequencies can be used, or word frequencies can be computed based on likely inputs to the BOT script. In the present implementation, base-level Recall conditions (conditions that test whether a boolean attribute is set for the given user record) are arbitrarily assigned a frequency of 0.25 (since it is difficult in general to compute the likelihood of an attribute being set), while base-level matching conditions are assigned a frequency based on the frequency of words in the Example statements found in the BOT script, since these Example statements are intended to represent a reasonable sample of the inputs that the BOT script is expected to handle. Example statements are additionally used in the validation and verification of operation of a virtual BOT, as is more fully discussed below.

If a matching condition is testing an input for a particular word, the frequency of that condition is the frequency of that word within the set of Examples. If it is testing an input for a partial word (such as a word beginning with the string “develop”), the frequency is the combined frequency of all words in the set of Example that match the partial word. If it is testing an input for a string of words, the frequency is the product of the frequencies of the individual words; it will be appreciated that other metrics, such as a true count of the frequencies of word pairs, could be used instead.

In the present embodiment, compound conditions (conditions composed of disjunctions or conjunctions of base-level conditions) are assigned specificity values based on the specificity of their child conditions. The specificity of a disjunction of one or more conditions is equal to

the highest specificity values from among the true children, while the specificity of a conjunction of one or more conditions is equal to the sum of the specificity values of all children, reduced by a fixed constant (currently 1000) for each child beyond the first, reflecting the fact that conditions tested together tend to be correlated. It will be appreciated that computing the specificity of conjunctions or disjunctions can be accomplished in a myriad of variations, the only requirement being that the essential meaning of 'and' and 'or' be preserved such that compound conditions composed of conjunctions return a value at least as high as the value returned for a similar compound condition composed of disjunctions.

A similar method is used in computing the specificity of conditions that involve PatternLists or optional elements. The matching of a PatternList has the specificity of the most specific element of the list that actually matched the input, while the specificity of an optional element or condition is zero if it is not true, and its normal specificity if it is. Therefore, the BOT author can include optional conditions that do not affect whether a condition is matched, but make it more likely that the condition will be chosen if the optional condition is true.

C. COMPILATION OF THE CATEGORY SELECTION TABLE

The first step in the compilation process is to compile the BOT script into executable structures, using standard compilation techniques well-known to those skilled in the art. Once the set of executable categories has been created, the standard categories are further compiled into a data structure (**CBFMatcher**) that is used to compute the set of active categories and select among them.

Once the categories have been compiled, the compiler identifies all "base-level blocks" in the categories. We define a "base-level block" as an IF block that contains at least one non-IF statement at its top level; it may contain other IF blocks as well. Normally, each base-level block corresponds to an action (**Say** or **Do**) taken by the BOT, although it might instead be some other statement such as **Remember**. In the code fragment below, the block beginning *IfHeard "bot"* and the block beginning *IfDontRecall* are both base-level blocks, as is the block beginning *If ?WhatUserSaid Matches "are*"*, which contains a **SwitchTo** statement at its top level. The block beginning *IfHeard "you"* is not a base-level block, as it contains no statements other than another **If** block.

```

Topic "Are you a bot" is
  IfHeard "you" Then
    If ?WhatUserSaid Matches "Are *" Then
      IfHeard "bot" Then

```

```

        Say "Yes, I am a bot";
        Done
        SwitchTo "Are you X";
        // The following will be executed if "Are you X"
5      // executes a SwitchBack
        IfDontRecall ?AnsweredQuestion Then
            Say "I don't know";
        Done
        Continue
10    Continue
    EndTopic

```

An IF block is said to be "active" if its condition is true and the conditions of all IF
 blocks containing that block are also true; in other words, the statements in the block would be
 15 executed if the category containing the block were executed in the standard fashion, and no
 Done, NextTopic, or SwitchTo statements were executed earlier in the block.

Each block is made true by some condition, which can be expressed in terms of
 disjunctions and conjunctions of base-level conditions, where each base-level condition is either
 the testing of a memory attribute (IfRecall or IfDontRecall) or a single pattern-matching
 20 operation. For the purposes of category selection, IfChance conditions are treated as always true
 and thus are not included in the condition set.

Figure 9 illustrates the data structure that is used to compute the set of active categories
 and select among them. Best-Fit Matcher 900 consists of a set of standard base-level blocks 902,
 a set of base-level blocks 904 that must be evaluated at run time, a set of base-level blocks 906
 25 that contain only negated conditions, a set of matching objects 908 that determine whether
 conditions are true or false, statistics 910 about the frequency of words in the domain, and a large
 number of other temporary variables 912 used in compilation and execution. The distinction
 among the sets of blocks 902, 904, 906 will be discussed below.

Figure 10 expands the view of a block 902, 904, 906. Each such block corresponds to a
 30 base-level block in the BOT script. Each block consists of a pointer 1002 to the category in
 which it is found in the script, a list 1004 of conditions 1006 that must be true in order for the
 block to be active, and a negation flag 1008 for each such condition. In the present
 implementation of the invention, conditions are always non-negated, and negation of conditions
 is implemented as a flag in the block containing the conditions rather than in the conditions
 35 themselves for efficiency reasons. Other implementations of negation of conditions are obvious
 to those skilled in the art.

The set of blocks 902, 904, and 906 corresponds to a subset of the base-level blocks in
 the BOT script. Given that the mechanism ultimately selects categories rather than blocks, it

excludes from consideration any blocks that might be active at the time of category selection but may not be active when the category is actually executed. Such blocks can exist because an earlier statement within the category (such as a Remember or Do statement) may change the BOT state, thus causing the block to no longer be active. For instance, in the code fragment shown above, the block beginning *IfDontRecall* follows a **SwitchTo** statement, and is thus not considered as a possible activator for the topic "Are you a bot". The topic "Are you X" might not execute a **SwitchBack** command, or might change the value of the flag *?AnsweredQuestion*, thus causing the **IfDontRecall** condition to have a different value at run-time than at category-selection time. Note that the block will only be executed if the block that contains it (the block beginning *If?WhatUserSaid Matches*) is active, so there is still a base-level condition that will activate the category whenever the *Say "I don't know"* action would be taken during category execution.

Therefore, any block following a non-IF statement guaranteed to be executed in the same category but before that block is excluded from consideration. The behavior of the mechanism might be improved by testing such statements to determine whether they can in fact affect the conditions in the block. Furthermore, when the set of active blocks is computed at run-time, any block other than the first active block in a particular category is excluded from consideration, as blocks other than the first active block can not be guaranteed to be executed. Therefore, blocks that follow a block consisting only of an Always or IfChance condition can also be excluded from consideration at compile time, as blocks consisting only of Always or IfChance conditions are always active.

Each condition 1006 corresponds to a condition found in the BOT script. There are two general classes of conditions; *BaseLevelConditions* and *IntermediateLevelConditions*. **Figure 11** illustrates a base-level condition. A base-level condition 1100 represents a single condition in the BOT script, either the Recall (or DontRecall) of a single user attribute or the pattern match of a single object to another object. In the present implementation, Chance conditions are considered to be always true for the purpose of category selection, and are therefore not converted into *BaseLevelConditions*.

In the present implementation of the mechanism, low-level conditions are often used to activate higher-level conditions that depend on them, or to activate blocks themselves. Therefore, each *BaseLevelCondition* includes pointers 1102 to any *Blocks* 902 that it activates, and pointers 1104 to any *IntermediateLevelConditions* 1106 that it activates. The circumstances under which a *BaseLevelCondition* activates a *Block* 902 or *IntermediateLevelCondition* 1106 are disclosed below.

Ordinarily, the value of a condition is set by a matching object described below. This value is assigned by setting a flag 1108. However, in the present implementation of the invention, there are some conditions whose values, due to efficiency reasons, are not set by an external matching object. Such conditions must also include a “run-time matching object” 1110 that can be called when necessary to determine a value for the condition; other conditions merely contain a NULL pointer in place of matching object 1110. The representation and implementation of such run-time objects would be well known to one skilled in the art.

As stated earlier, all of the BaseLevelConditions are non-negated conditions; that is, they are true if an attribute is recalled or a matching operation succeeds. If the actual condition found in the BOT is a negated condition, this information is included in the MatcherBlock or IntermediateLevelCondition that contains the BaseLevelCondition. However, BaseLevelConditions can be “optional” if they represent a condition that is not needed in order for a block to be active, but, if true, should be considered when selecting the block that most specifically matches the input. Such optional conditions are indicated through the use of a flag 1112.

Figure 12 illustrates an intermediate-level condition. An intermediate-level condition represents the conjunction or disjunction of one or more conditions, either base-level or intermediate-level, known as “child conditions.” (An intermediate-level condition with only a single child condition could clearly be replaced by its child condition.) The intermediate-level condition structure 1106 includes a list 1200 of child conditions 1202. As in Matcher Blocks, the child conditions can be negated; such negated child conditions are indicated by a negation flag 1204. A flag 1206 indicates whether the condition itself is a conjunction or disjunction of the child conditions. Finally, each condition can activate a set 1102 of matcher blocks 902 and a pointer 1208 to at most one other IntermediateLevelCondition 1106. The circumstances under which an IntermediateLevelCondition activates a Block 902 or IntermediateLevelCondition 1106 are disclosed below.

In the present implementation, these conditions correspond closely to the conditions and objects built in the compilation of the BOT script; the matching of a single PatternList is a BaseLevelCondition, while the matching of a list of objects (separated by commas in the BOT script) is represented as an IntermediateLevelCondition disjunction. In the present implementation, IntermediateLevelConditions form a forest of conditions; BaseLevelConditions may be the children of many IntermediateLevelConditions, but IntermediateLevelConditions have only one parent condition of which they are a child. There are many logically equivalent

ways in which a BOT script can be represented as IntermediateLevelConditions, and the present invention should not be limited to any particular method for such representation.

Figure 13 illustrates the structure of CMatcherPropertyTester objects 908, which serves to set the values of most of the BaseLevelConditions 1100. Nearly all of the conditions found in most BOT scripts in our current scripting language consist of either testing the recall of a memory attribute or matching the value of a memory attribute (such as the system-defined attribute ?WhatUserSaid) against a fixed pattern or PatternList. Therefore, for each memory attribute that is used in a condition in some standard category in the BOT script, the truth value of certain BaseLevelConditions 1100 will need to be set. Each such memory attribute has a PropertyTester 908 specific to that attribute name 1300. Such a property tester includes a pointer 1302 to the BaseLevelCondition 1100, if any, for the recall of that attribute. It also includes a pointer 1304 to the BaseLevelCondition 1100, if any, that corresponds to a condition testing the optional recall of that attribute (such a condition is always true, but may affect the selection of a category as described below.) Finally, the PropertyTester 908 will perform pattern matching for any BaseLevelCondition 1100 that tests that attribute 1300 against a fixed pattern. Matches and Contains tests (and their negations) are handled using a PatternMatching structure, indicated by a pointer 1306 to the first node of a PatternMatcherObject for that attribute (more details on this pattern matching process are disclosed below.) ExactlyMatches tests that compare the attribute to a fixed pattern can be computed efficiently without requiring any pattern-matching, by simply using a map 1308 from fixed pattern strings to BaseLevelConditions 1100.

Any BaseLevelCondition 1100 that is not included in any PropertyTester 908, either because the left-hand side of the match is not an attribute or because the right-hand side is not a fixed pattern, is labeled as a run-time condition and assigned a run-time matching object 1110. Such a condition is individually tested when needed at run-time; clearly it is undesirable from an efficiency standpoint to use too many of these, but any arbitrary condition can be stored as a RunTimeCondition if necessary.

Figure 16 illustrates a block/condition structure created from a particular BOT script to be described in the following section. The details of **Figure 16** are discussed in the following section, but the overall structure of the example illustrates the structure of blocks and conditions. The blocks 1600, 1602 are base-level blocks in the BOTscript. Conditions 1608, 1610, 1612, 1614, 1616, 1618 are base-level conditions 1100 whose values are set by PropertyTesters 908 (not shown in **Figure 16**). Conditions 1620, 1622, 1624, 1626 are intermediate-level conditions 1106. The heavy arcs 1604, 1636 leading to the blocks 1600, 1602 correspond to block activation pointers 1102 in the conditions. The heavy arcs 1628, 1634, 1638 between base-level

conditions and intermediate-level conditions represent condition activation pointers 1104, while the heavy arcs 1630, 1632 between intermediate-level conditions represent parent activation pointers 1208. All arcs directly below an intermediate-level condition represent child pointers 1200.

5 Once the values of the BaseLevelConditions 1100 are determined (except for the run-time subset of the base-level conditions), the value of any IntermediateLevelCondition 1106 can, when needed, be computed in top-down recursive fashion, using child pointers 1200 and recursive computation techniques well-known to those skilled in the art. In this process, the values of BaseLevelConditions and RunTimeConditions are tested or computed as needed. In
10 the structure displayed in **Figure 16**, the activation value of the block 1600 would be determined by computing the value of the intermediate-level condition 1624 and testing the value of the base-level condition 1616; if both were true, the block would be active. The value of intermediate-level condition 1624 would be determined by computing the values of its child intermediate-level conditions 1620 and 1622; if one or both are true, then intermediate-level
15 condition 1624 is true. ("Short-circuiting" evaluation could be used to avoid computing the second condition if the first is true, but cannot be used if the specificity values of the conditions will be needed later.) Similarly, the value of condition 1620 is determined from the values of base-level conditions 1608 and 1612, while the value of condition 1622 is determined from the values of base-level condition 1610 and 1614. (Recall that the values of all base-level conditions
20 were determined earlier.)

 However, the fact that under ordinary circumstances only a small subset of the BaseLevelConditions are true can be used to compute the values of the IntermediateLevelConditions much more efficiently than by using the top-down method described above. In the present implementation of the invention, the first step in the selection of
25 a category given an input is to use all of the PropertyTester objects 908 to determine the subset of the ordinary BaseLevelConditions 1100 that are true given the input. Any BaseLevelCondition, other than the RunTimeConditions, that is not labeled as true by this process can be assumed to be false. Since in any particular situation there are generally far more conditions in a BOT script that are false than are true, this process is significantly more efficient
30 than testing each BaseLevelCondition individually. Thus, after the PropertyTesters are executed, there are three sets of BaseLevelConditions: those known to be true, those known to be false, and the RunTimeConditions, for which no value has yet been determined.

 The values of the IntermediateLevelConditions are determined in a similar "bottom-up" fashion, in which any condition that is not directly labeled as true is known to be false. For each

true BaseLevelCondition 1100, the list 1104 of IntermediateLevelConditions 1106 are marked as activated. If the IntermediateLevelCondition 1106 is a disjunctive condition, it is known to be true once a BaseLevelCondition activates it. If the IntermediateLevelCondition 1106 is a conjunctive condition, the children 1200 (other than the activator BaseLevelCondition, which is already known to be true) must be tested. For any non-run-time BaseLevelConditions among the children, the truth value will already have been computed. For any run-time BaseLevelConditions among the children, the run-time matcher object 1110 is called (or potentially, a cached value may be used instead if the matcher object 1110 has already been evaluated.) For any IntermediateLevelConditions among the children, the top-down evaluation algorithm described above is used. Thus, the truth value for the IntermediateLevelCondition 1106 is computed. If the condition 1106 is true, and its parent activation pointer 1208 is non-null, its parent condition is activated, and the process is repeated.

Meanwhile, for any true BaseLevelCondition 1100 or true IntermediateLevelCondition 1106 that is encountered in this process, the list 1102 of blocks 902 is also activated. Since blocks are conjunctive conditions, if the activated block 902 has more than one child condition 1006, the other children are evaluated using the top-down evaluation procedure described above.

In order to prevent this algorithm from repeatedly visiting conjunctive IntermediateLevelConditions, each conjunctive condition has one of its child conditions selected as "activator"; this child condition is marked so that it activates the parent node. (Recall that IntermediateLevelConditions are the child of only one condition, while BaseLevelConditions may be the child of many conditions. Therefore, if the child is a BaseLevelCondition 1100, a pointer to the parent is placed in the list 1104 of conditions, while if the child is another IntermediateLevelCondition 1106, the pointer to the parent is placed in the parent pointer 1208.) Blocks are handled similarly to conjunctive conditions, although since a condition may be shared among several blocks (if the blocks are within a single higher-level **if** block within the BOT script), both BaseLevelConditions and IntermediateLevelConditions may activate a list 1102 of blocks 902. Like conjunctive conditions, blocks have only a single activator.

Disjunctive conditions can be activated by any of their children, and thus all children of a disjunctive condition have activator pointers to their parent condition.

To return to the example in **Figure 16**, note that some of the arcs (e.g. arc 1628) are shown with heavy lines. These heavy arcs represent activator pointers. Thus, if BaseLevelCondition 1612 is found to be true, it will activate its parent IntermediateLevelCondition 1620. The selection mechanism will then check the other children of condition 1620 (in this case, BaseLevelCondition 1608.) If they are all found to be true,

condition 1620 will be true, and will activate IntermediateLevelCondition 1624, which will in turn activate MatcherBlock 1600. The other child of the MatcherBlock, BaseLevelCondition 1616, will now be tested; if it is true the MatcherBlock will be true. Note that the other BaseLevelConditions 1608 and 1616 used in the computation are not activator nodes and thus do not activate any IntermediateLevelCondition. If these two conditions 1608 and 1616 were true but the activator condition 1612 were not, no IntermediateLevelConditions would be activated and no computation would be done for this block.

In order for this activation method to work efficiently, no IntermediateLevelCondition or MatcherBlock should be activated by a condition whose value is false or unknown. Recall that the number of true conditions is generally far smaller than the number of false conditions. If the activator for a condition or block is a negated child, the condition or block will be activated when the child is false, which will occur most of the time. Therefore, negated children are never chosen as activators. If instead the activator is a RunTimeCondition, its value will be unknown at run-time and must be computed every time the selection process is run. Therefore, RunTimeConditions are also never selected as activators. Finally, if a child of a disjunctive IntermediateLevelCondition is inappropriate as an activator, for instance because it is negated or depends on a RunTimeCondition, the disjunctive condition will be inappropriate as an activator and should not be chosen as an activator for other IntermediateLevelConditions or MatcherBlocks. The selection of activators for IntermediateLevelConditions and MatcherBlocks is discussed in greater detail below.

This restriction leaves open the possibility that certain MatcherBlocks will have no suitable child conditions to be chosen as an activator, if all conditions involved in the block depend on RunTimeConditions or negated conditions. Such blocks must be explicitly tested each time category selection is to be done, and are included in the matcher object 900 in the list 904 of non-activated matcher blocks. In addition, RunTimeConditions whose values depend on the testing of other conditions (in the present scripting language, this is exactly the set of conditions that include a reference to the value of a wildcard from an earlier match, such as a *1 or #1 value) are implemented by requiring that the conditions in the entire block be tested at run-time in order to determine the value of the reference. Such blocks are stored in the matcher object 900 in the list 906 of run-time matcher blocks.

The mechanism for category selection will function regardless of the method used for selection of an activator for each MatcherBlock and conjunctive IntermediateLevelCondition. However, the mechanism will function most efficiently if these activators are chosen such that (1) every IntermediateLevelCondition and MatcherBlock is activated as infrequently as possible,

and (2) activators that require additional computation beyond that done by the PropertyTesters are not used.

Therefore, activators are chosen according to the frequency with which they are likely to be true. In the present implementation, the frequency estimate of Recall conditions and word matching is the same frequency value used in computing run-time specificity, based on the frequency of words in the BOT script. The frequency estimates for PatternLists, optional elements, and compound conditions are computed somewhat differently than the run-time estimates, however. For purposes of activator selection, the frequency of a condition that tests for a PatternList is equal to the frequency of the most common element of that PatternList; it is also plausible to use other measures, such as the sum of the frequencies of all elements of the PatternList. BaseLevelConditions that are negated, optional, or depend on a RunTime computation are arbitrarily assigned a frequency of 1 in order to prevent them from being chosen as activators.

The frequency estimate for an IntermediateLevelCondition is computed based on the frequencies of its elements. For a disjunctive condition, the frequency of the condition is estimated to be the frequency of the most frequent child. Other formulas, such as the sum of the frequencies of all children, could be used instead.) For a conjunctive condition, the frequency of the condition is estimated to be the product of the frequencies of all children, multiplied by a constant (currently 2) for each child beyond the first, to represent the fact that the conditions that are tested in a conjunctive condition in a BOT script tend to be correlated. It will be appreciated that the mechanism will function efficiently as long as the frequency estimates are approximately correlated with the true frequencies of the conditions; it is not necessary for the frequency estimates to be close to the true frequencies, nor to be computed using the particular formulas described above.

Once the frequency has been estimated for each node, the activators for each node are computed by selecting the child with the lowest frequency value for each MatcherBlock or conjunctive IntermediateLevelCondition. This will result in the conditions and blocks being activated as infrequently as possible, while preventing any negated conditions or RunTimeConditions from being chosen (as those conditions are assigned a frequency of 1.) Any MatcherBlock for which all children have frequency value 1 is not assigned an activator and is stored in the list 904 of non-activated MatcherBlocks.

D. USE OF THE CATEGORY SELECTION TABLE

The pattern-matching system of the PropertyTesters will now be discussed in greater detail. The function of the pattern-matching system is to map attribute values (such as the input value ?WhatUserSaid) into the set of pattern-matching BaseLevelConditions that are true given the value. The implementation of the pattern-matching system can be viewed as non-

5 deterministic finite state automata (NFA) where each node represents a pattern and each arc represents an element for pattern matching, for instance a word, space, punctuation mark, wildcard character, etc. **Figure 15** illustrates an NFA pattern-matcher created from a particular BOT script, to be discussed later.

Figure 14 displays the structure of a single node in a pattern-matching NFA. Each node

10 1306 is associated with a partial pattern 1400 that represents some initial substring (or possibly the entire string) of at least one pattern found in the BOT script. When the node is active, this indicates that the input to the pattern-matching NFA, as processed so far, matches the pattern 1400. This pattern value 1400 is included in the representation only for debugging and explanation purposes; it is not used in the actual execution. If the pattern 1400 corresponds to a

15 pattern matching condition that is found in the BOT script, there is a pointer to the BaseLevelCondition 1100 that corresponds to that pattern matching operation. (Otherwise this pointer is NULL.) If the node is active when the end of the input is reached, the BaseLevelCondition 1100 will be marked as being true.

Words, spaces, punctuation marks, one-word wildcard characters, and similar objects that

20 match a fixed element of the input are represented as arcs in the NFA. These arcs are represented in a pattern-matcher node 1306 as a map 1402 of associations 1404 of such objects with pointers to other nodes. For example, arc 1542 (representing the word "you") and arc 1568 (representing a space) are such arcs.

Optional elements are represented by having the element itself be represented by one or

25 more normal arcs 1404 and adding an epsilon-move (an arc that can be taken without processing any input symbol) between the node immediately preceding the optional element and the one at its end. This set of all optional-element epsilon-move arcs 1412 from a node 1306 is stored as a list 1410 within the node. (Note that the category selection mechanism, as described, allows for optional elements both within the pattern-matching structure and at the level of

30 BaseLevelConditions. Optional elements within the pattern-matching structure are less efficient but are needed in the case of optional elements within concatenations.) For example, unlabeled arc 1558 is such an epsilon-move arc.

PatternLists are represented with a normal arc 1404 for each element of the pattern list, and an epsilon-move arc 1408 between the end node for each element of the pattern list and

another node representing that the PatternList itself has been observed. Such epsilon-move arcs 1408 are stored in the appropriate node 1306 in a list 1406. This representation for PatternLists is used in order to avoid the generation of an exponential number of nodes in the case where two or more PatternLists are concatenated. For example, unlabeled arc 1550 is such an epsilon-move arc.

Finally, true wildcard characters that can match zero words or many words are represented an arc 1414 from the node 1306 (preceding the wildcard) to another node. This wildcard node contains arcs for any patterns that can follow the wildcard, and also contains an arc that loops to itself on any input. This implementation is needed since other arcs 1404 may leave the original node 1306 that do not allow any other input between the pattern 1400 and the label on the outgoing arc 1404. If there are no such arcs 1404, the extra node can be combined with the original node 1306 and the self-looping arc can be included in the node 1306. All of the wildcard arcs in **Figure 15** have been combined in this way. **Figure 17** illustrates the original (non-combined) form of some of the arcs in **Figure 15**.

The techniques used to create an NFA from a set of patterns to be matched are well-known to those skilled in the art. For each condition in the BOT script that compares an attribute value to a pattern using "Contains", "Matches", or their negations, a path is added from the start node in the PropertyTester for that attribute to the node corresponding to that pattern, and a link is added from that node to the corresponding BaseLevelCondition 1100. A Matches condition is directly represented as a path in the NFA; a Contains condition is represented by including a * wildcard at the beginning and end of the path.

Conditions that use ExactlyMatches as a test are very efficient to test, so they are tested by including a hash table 1308 in the PropertyTester 908 rather than by using the NFA. As described above, conditions for which the left-hand-side of the comparison is a fixed string or star-buffer value are not computed using BaseLevelConditions at all.

Given the NFA representation of a set of patterns, such as that shown in **Figure 15**, determining the patterns that are matched by a particular input can be done by applying the NFA to the input. The techniques for executing a NFA are well known to those skilled in the art. In the present implementation of the invention, the start node of the NFA corresponds to the node matching an empty pattern. There is no single terminal node for the NFA; instead, any node that corresponds to a pattern that is actually used in the BOT script has a pointer to the appropriate BaseLevelCondition. Once the input has been completely processed by the NFA, each node that is active will activate the BaseLevelCondition, if any, that is associated with that node. The execution of the NFA shown in **Figure 15** on several inputs is discussed in the following section.

As discussed above, each BaseLevelCondition 1100 may be an activator for one or more IntermediateLevelConditions 1106. Each true BaseLevelCondition 1100 activates each of the IntermediateLevelConditions 1106 in its list 1104 of conditions. If a disjunctive IntermediateLevelCondition is activated, it is guaranteed to be true; therefore, if it contains an activation pointer 1208 to another IntermediateLevelCondition, that condition is activated (the disjunctive condition is also flagged as being true so that if it is later activated by another child, the activation process does not need to be repeated.) If a conjunctive IntermediateLevelCondition is activated, all the other children of the node are evaluated, and if all are true, the parent condition 1208, if any, is activated. (Recall that only one child of a conjunctive IntermediateLevelCondition is selected as an activator, so this process will not be repeated for other children of the same node.) Finally, both BaseLevelConditions and IntermediateLevelConditions can be activators for MatcherBlocks. For each condition found to be true during the above process, each block 902 in the list 1102 of MatcherBlocks in the condition is activated. As with conjunctive IntermediateLevelConditions, once a MatcherBlock 902 is activated, the other conditions 1006 in the block must be tested using the top-down evaluation process described earlier. If all of the conditions associated with the MatcherBlock 902 are true, the MatcherBlock is selected as an active block. Finally, those MatcherBlocks 904, 906 that do not have an activator condition (as discussed earlier) must be tested explicitly by performing top-down evaluation of the conditions in each block.

For a given input, there will often be more than one active block. Unless the blocks are explicitly given a fixed ordering in the script, the BOT execution mechanism needs an algorithm to select among them. In the present implementation, the block chosen is the one for which the specificity value is highest. Blocks in categories that have already been executed are excluded from the computation. In the present implementation, if more than one block is active in the same category, only the first of those blocks is eligible for selection, as the actual execution of the category will execute the entire category, and thus is guaranteed to execute the first active block. (Note that it is not guaranteed to execute later blocks in the category even if they are selected as active, as the first block may terminate execution, or may change the state of the BOT.) Clearly the mechanism could instead consider all active blocks within a category as possibilities.

These specificity values could be pre-computed in a manner similar to that used when selecting the activators for IntermediateLevelConditions and MatcherBlocks. However, in cases where a block includes disjunctive conditions, it is possible to make a more accurate run-time estimate of the frequency of the actual input. For instance, a particular block might respond to

both the question “Who is Walter?” and the question “Who is he?”. The first question is clearly a more specific question than the second, as can be estimated by comparing the frequency of the words “Walter” and “he”. Thus, in order to get the best possible performance from the selection mechanism, the specificity estimates for each active block may need to be computed at run time.

5 (Although this computation is described as a separate process in this description, the computation is actually done simultaneously with the selection of the active blocks.)

The run-time estimate of the specificity value for a BaseLevelCondition is based on the set of paths that were taken from the start node of the matcher NFA to the node that corresponds to the BaseLevelCondition. If there is only one such path for the given input, the specificity
10 estimate is simply the sum of the specificity of all the arcs in the path. (In fact, if there is only one possible path between the start node and the condition node, the specificity value can simply be computed at compile time. If there is more than one possible path, the specificity estimate may vary and must be computed at run-time.) If there is more than one path for the given input (either because there is an optional element that was found, or because two or more elements of a
15 PatternList were found), the highest specificity value is chosen. (It will be appreciated that other heuristics may also be used.) In the present implementation, the specificity values for arcs are based on the frequency of words in the Examples found in the BOT script, but other sources of frequency estimates could also be used, including the possibility of keeping a database of word frequency in the questions that are asked to the BOT as it is executing. The following section
20 includes several examples of specificity computations for various inputs based on the NFA in **Figure 15**.

Similarly, the specificity values for IntermediateLevelConditions are also computed at run-time based on the BaseLevelConditions that were activated. For each disjunctive IntermediateLevelCondition, the specificity value is simply the highest specificity value from all
25 true children of the condition. For a conjunctive condition, as above, the specificity value is the sum of the specificity values of all children, reduced by a constant value (currently 1000) for each child beyond the first. The specificity value of the MatcherBlocks is estimated in exactly the same way as the specificity value of a conjunctive IntermediateLevelCondition. Negated conditions have a fixed specificity, although they could be assigned a specificity value based on
30 1 minus the frequency of the unnegated condition. Completely optional conditions have a specificity of 0 if they are not true, and their true specificity value if they are. Conditions that are only computed at run-time can be assigned specificity values based on the frequencies of the words in the input that actually match the condition. (Again, it will be appreciated that a variety of methods could be used to estimate the frequency of a condition at run-time, including the

direct use of an estimate generated at compile-time or the assignment of frequency values by the BOT author, and that the scope of the present invention should not be limited to any particular method of estimating condition frequency.) The following section includes several examples of specificity computations for various inputs based on the condition structure in **Figure 16**.

5 Once the specificity value for each active block has been computed, the activation mechanism simply selects the block with the highest specificity value for execution, breaking ties according to the Focus of Attention mechanism as discussed above. The category containing this block is then executed in the usual way. The remaining blocks can also be stored if the user is attempting to debug the BOT script, as the BOT author may need to know all of the
10 blocks that were activated by a particular input. If the category returns a value of Continue or NextTopic, the process can be repeated as long as it finds one or more active blocks that have not already been executed.

E. EXAMPLES OF AUTOMATIC SELECTION OF RESPONSE

15 Our first example demonstrates the compilation of a pattern-matching structure from a set of blocks. We begin with the following topic from a BOT script in our present scripting language:

```

20       PatternList BOTS is "bot", "virtual robot";
      Topic "Are you a bot" is
          // ?FactQuestion is a flag that is set for yes/no questions such as "Are you..."
          If Recall ?FactQuestion and Heard "you*" + BOTS Then
              Example "Are you a bot?";
              Say "Yes, I am a bot";
25       Done
      EndTopic
      Topic "Are you a sales bot" is
          If Recall ?FactQuestion and Heard "you*sales bot" Then
              Example "Are you a sales bot?";
30       Say "No, I am a FAQ bot";
      Done
      EndTopic
      // Note the {} characters which indicate an optional element.
      Topic "Are you a complex bot" is
35       If Recall ?FactQuestion and Heard "you*complex" + {BOTS} Then
          Example "Are you a complex bot?";
          Say "No, I'm a very simple bot";
      Done
      EndTopic
40
```

Figure 15 shows the pattern-matching structure created from the above script by the BOT script compiler. BaseLevelCondition 1500 corresponds to the pattern matching condition in the topic “Are you a bot”, BaseLevelCondition 1502 corresponds to the pattern matching condition in the topic “Are you a sales bot”, and BaseLevelCondition 1504 corresponds to the pattern matching condition in the topic “Are you a complex bot”. Circular nodes (e.g. start node 1506) correspond to nodes in the pattern matcher object; labeled arcs (e.g. arc 1542) correspond to transitions, unlabeled arcs (e.g. arc 1550) correspond to epsilon-moves, and dark-line arcs (e.g. activation arc 1572) correspond to arcs by which pattern matcher nodes activate BaseLevelConditions. The BaseLevelConditions corresponding to the IfRecall questions in the topics are not shown in **Figure 15**, nor are the MatcherBlocks themselves. In addition, the wildcard arcs (e.g. arc 1540) have been simplified by combining the two nodes that would otherwise be generated, as the script above does not result in any nodes which contain both wildcard and non-wildcard follow-ups. This combination of nodes is discussed below in the explanation of **Figure 17**.

As a first example, consider the pattern matching condition in the topic “Are you a sales bot”, which tests for the pattern “you*sales bot”. Since the condition is a “Contains” condition, it is represented in the pattern matcher as the pattern “*you*sales bot*”. This condition is compiled into the pattern matcher starting from the start node 1506. The first “*” is the label for the self-looping arc 1540 from start node 1506 to itself; the pattern “you” is the label for arc 1542 from node 1506 to node 1508; the “*” following it is the label for the self-looping arc 1544 from node 1508 for itself; the word “sales” is the label for arc leading from node 1508 to node 1510; the space between “sales” and “bot” is the label for the arc leading to node 1512; the word “bot” is the label for the arc leading to node 1514, and the final “*” is the label for the self-looping arc from node 1514 to itself. Thus, start node 1506 corresponds to a state in which a “*” has been matched (i.e. any input has been seen); node 1508 corresponds to the pattern “*you”; node 1510 corresponds to the pattern “*you*sales”, and so on. Node 1514 corresponds to the entire pattern “*you*sales bot*” and thus is connected via an activator link to BaseLevelCondition 1502.

Now examine the compilation of the pattern “you*” + BOTS from the topic “Are you a bot”. Since it is a “Contains” condition, the pattern is converted to “*you*” + BOTS + “*”. The first three elements of the pattern are represented as before, using arcs 1540, 1542, and 1544. At node 1508, additional arcs 1546 and 1548 are labeled with the words “bot” and “virtual”, respectively, representing the two elements of the PatternList BOTS that can follow node 1508. Since the word “bot”, if found, is a complete element of the PatternList, there is an epsilon-move

transition 1550 from node 1516 (corresponding to the pattern “*you*bot”) to node 1524 (corresponding to “*you*” + BOTS). In order to match the PatternList BOTS, the word “virtual” must be followed by a space (to node 1520) and the word “robot” (to node 1522); another epsilon-move transition 1552 leads from node 1522 to the “*you*” + BOTS node 1524. Finally, another * wildcard is used as the label for the arc from node 1524 to itself. Node 1524 corresponds to the complete pattern and thus activates BaseLevelCondition 1500 via activator link 1572.

Finally, the compilation of the pattern “you*complex” + {BOTS} differs from the previous pattern in two ways (other than the obvious inclusion of the pattern “complex” on arc 1554 leading to node 1526.) First, there is a Space label on arc 1556 that leads from node 1526 (“*you*complex”) to the node 1528 that represents the beginning of the patterns for PatternList BOTS. Arc 1556 represents the space that must separate the word “complex” from the word “bot” or “virtual” if the pattern is to successfully match; in the present embodiment of the scripting language, the “+” operator produces an implicit space between words if there is no other wildcard (such as a “*”) that separates them. (Note that the paths between node 1508 and nodes 1516 and 1518 did not need to include an extra space arc, as the space in the input would be matched by the * on arc 1544; the compiler optimizes the pattern matching structure by not including the extra space.) The PatternList BOTS is represented by nodes 1530, 1532, 1534, 1536, and 1538, corresponding exactly to nodes 1516, 1518, 1520, 1522, and 1524 respectively. Finally, there is an epsilon move transition 1558 from node 1526 to node 1538 representing the fact that the entire PatternList BOTS is an optional element in matching. Another * wildcard labels the arc from node 1538 to itself, and node 1538 contains an activation arc that activates BaseLevelCondition 1504.

Now the execution of the structure shown in **Figure 15** on various inputs will be demonstrated. For purposes of illustration in the following example, assume that the specificity of various words has been computed or assigned as follows:

You	3000
Bot	4000
Virtual	8000
Robot	8000
Sales	6000
Complex	8000

Suppose the matcher is given the input “Are you a bot”. As processing of the input begins, the only active node is the start node 1506. The word “are” matches only the arc 1540

labeled "*", so 1506 remains the only active node. The space between "are" and "you" has the same result. The word "you" matches both arc 1540 labeled "*" and arc 1542 labeled "you", so both node 1506 and node 1508 are active. The next space, the word "a", and the space following "a" all match only the "*" arcs 1540 and 1544, so nodes 1506 and 1508 remain the only active nodes. The word "bot" matches the "*" arcs 1540 and 1542 and the arc 1546 labeled "bot", so nodes 1506, 1508, and 1516 are now active. (None of the other arcs leaving node 1508 represent possible transitions given the word "bot".) The epsilon-move arc 1550 also causes node 1524 to become active without any input processing. The matcher has now reached the end of the input string, and nodes 1506, 1508, 1516, and 1542 are active. The only one of these which activates a BaseLevelCondition is 1542, so BaseLevelCondition 1500 is activated. This activation of BaseLevelCondition 1500 is assigned a specificity of 7000 based on the specificity values of the labels of the arcs 1540, 1542, 1544, 1546, and 1550 followed on the path from the start node to the activating node 1542. (The * labels on arcs 1540 and 1544 do not have any specificity value.) Eventually, this BaseLevelCondition will activate a MatcherBlock corresponding to the condition in the topic "Are you a bot". In the present embodiment of the invention, the specificity value of a BaseLevelCondition is computed as the sum of the specificity values assigned to all arcs on the path to the activating node. However, it would be obvious to one skilled in the art that other methods of combining specificity values would be possible, such as simply taking the maximum specificity value of any arc along the path, and that the scope of the present invention should not be limited to any particular method of such combination.

Now suppose the matcher is given the input "Are you a sales bot?". As before, the first part of the input, "Are you a ", activates nodes 1506 and 1508. The word "sales" matches arc 1566 (labeled "sales") as well as arcs 1540 and 1544 (labeled "*"), so nodes 1506, 1508, and 1510 are now active. The space following the word "sales" matches arc 1568 as well as the "*" arcs, so nodes 1506, 1508, and 1512 are now active. (Node 1510 is no longer active as no arc leading to 1510 was matched by the space.) The word "bot" matches arcs 1546 (from node 1508) and 1570 (from node 1512) as well as the "*" arcs 1540 and 1544, so nodes 1516 and 1514 are now active as well as nodes 1506 and 1508. (Node 1512 is no longer active.) The epsilon-move arc 1550 also activates node 1524. The final "?" is now processed; nodes 1506, 1508, 1514, and 1524 all have "*" arcs leaving them, so all remain active, while node 1516 is no longer active. The matcher has now reached the end of the input string, so node 1514 and node 1524 activate BaseLevelConditions 1502 and 1500, respectively. As before, BaseLevelCondition 1500 is assigned a specificity of 7000 for this activation. BaseLevelCondition 1502 is assigned a specificity of 13000 based on the fact that the words

“you”, “sales”, and “bot” on arcs 1542, 1566, and 1570, respectively, were all matched on the path to the activator node 1514. (The space, like the * wildcards, does not add to the specificity in the present implementation.) Eventually, both the block in the topic “Are you a sales bot” and the block in the topic “Are you a bot” will be active, and the matcher will choose the block in the topic “Are you a sales bot” as it has a higher specificity value.

Finally, suppose the matcher is given the input “Are you a complex virtual robot” As before, the first part of the input, “Are you a ”, activates nodes 1506 and 1508. The word “complex” matches arc 1554 (labeled “complex”) as well as arcs 1540 and 1544, so nodes 1506, 1508, and 1526 are now active. The epsilon-move arc 1558 activates node 1538 as well. Next, the space following “complex” matches arc 1556, activating node 1528. Meanwhile, the “*” arcs from nodes 1506, 1508, and 1538 also match the space, so nodes 1506, 1508, and 1538 remain true. Next, the word “virtual” matches arc 1548 (from node 1508), activating node 1518, and arc 1560 (from node 1528), activating node 1532. (Nodes 1506, 1508, and 1538 all remain active for the remainder of the matching process.) The space now activates nodes 1520 and 1534, and the word “robot” activates node 1522 and node 1536. Epsilon-transition arcs now activate node 1524 (from node 1522) and repeat the activation of node 1538 (from node 1536). The matcher has now reached the end of the input string, so nodes 1524 and 1538 activate BaseLevelConditions 1500 and 1504, respectively. BaseLevelCondition 1500 is assigned a specificity of 19000 for this activation, based on the words “you”, “virtual”, and “robot” (note that it has a much higher specificity than in its previous activation.) Node 1538, the activator for BaseLevelCondition 1304, was activated by two separate paths. The path involving arcs 1542, 1554, and 1558 had a specificity of 11000, while the path involving arcs 1542, 1554, 1556, 1560, 1562, and 1564 has a specificity of 27000, based on the words “you”, “complex”, “virtual”, and “robot”. BaseLevelCondition 1504 is assigned the specificity of the most specific path to its activator node 1538, or 27000, and therefore will cause the topic “Are you a complex bot” to be selected ahead of the topic “Are you a bot”. (Note that if the optional element {BOTS} had not been included in the pattern in “Are you a complex bot?” the topic “Are you a bot” would have been selected instead.)

Figure 17 illustrates in more detail the operation of the wildcard arcs in the NFA shown in **Figure 15**. As discussed above, a * wildcard arc is actually implemented as an epsilon-move transition to another node, which contains a self-looping arc that matches any input. Thus, start node 1700 has an epsilon-move transition 1702 leading to an intermediate node 1704, which contains an arc 1706 (corresponding to * arc 1540 in **Figure 15**) that loops back to itself. The arc 1542 for the word “you” is thus a transition from intermediate node 1704 to node 1708 rather

than directly from start node 1506 to node 1508. The entire set of start node 1700, epsilon-move transition 1702, intermediate node 1704, and * arc 1706 are equivalent to start node 1506 and * arc 1540 in **Figure 15**.

This implementation of the * wildcard arcs is necessary in order to distinguish between the empty string and the wildcard pattern "*", as well as to distinguish between words separated by spaces and words separated by * wildcard characters. For instance, suppose the BOT script contained a condition :

If ?WhatUserSaid Matches "what".

In order to test for this condition, the arc labeled "what" would need to exit from node 1700 in **Figure 17**; if it exited from node 1506 in **Figure 15**, or from node 1704 in **Figure 17**, an input such as "so what" would be considered to match "what"; the word "so" and the space following it would match the wildcard character and the word "what" would match the arc labeled "what".

Similarly, the node 1508 and its self-looping * arc 1544 in **Figure 15** is represented by a node 1708, an epsilon-move transition 1710, an intermediate node 1712, and a * arc 1714. Thus, if an IfHeard condition in the BOT script contained a pattern such as "you are", the "space" transition would leave from node 1708 rather than from node 1712, thus preventing inputs such as "you sure are" from matching the pattern "you are".

A final example demonstrates the construction of IntermediateLevelConditions and the selection of activator nodes. We begin with the following two conditions, intended to answer questions such as "Are you expensive?", "What do you cost?", and "What does NeuroStudio cost?".

```
// Condition #1
IfHeard "you" Then
    If (Recall ?FactQuestion and Heard "expensive") or
       (Recall ?DescriptionQuestion and Heard "cost") Then
// Condition #2 (this would most likely be more complex in a real script)
IfHeard "cost" and "NeuroStudio" Then
```

It will be appreciated that "?FactQuestion" is a flag set for yes/no questions such as "Are...?" while "?DescriptionQuestion" is a flag set for "What..." questions. Both "?FactQuestion" and "?DescriptionQuestion" are flags that are set in a set of standard libraries that are being used for these examples.

Figure 16 illustrates the structure of BaseLevelConditions, IntermediateLevelConditions, and MatcherBlocks constructed by the compiler to handle the above conditions. (The

PropertyTester and PatternMatcher objects responsible for setting the values of the BaseLevelConditions are not shown in **Figure 16**.) MatcherBlock 1600 represents condition #1 above, and is the conjunction of IntermediateLevelCondition 1624 and BaseLevelCondition 1616 (the IfHeard “you” condition). IntermediateLevelCondition 1624 represents the disjunction of IntermediateLevelConditions 1620 and 1622, representing the two general question types covered by the condition. IntermediateLevelCondition 1620 is the conjunction of the recall BaseLevelCondition 1608 and the pattern matching BaseLevelCondition 1612, while IntermediateLevelCondition 1622 is the conjunction of the recall BaseLevelCondition 1610 and the pattern matching BaseLevelCondition 1614. MatcherBlock 1602 represents condition #2 above, and is the conjunction of IntermediateLevelCondition 1626 and the recall BaseLevelCondition 1610, while IntermediateLevelCondition 1626 is the conjunction of the pattern matching BaseLevelCondition 1614 and the pattern matching BaseLevelCondition 1618. The activator links for each non-base-level condition are shown as bold arcs, such as arc 1604, while the non-activator links are shown as plain arcs, such as arc 1606.

Now the execution of the structure shown in **Figure 16** on various inputs will be demonstrated. For purposes of illustration in the following example, assume that the specificity of various words has been computed or assigned as follows and that Recall conditions are assigned an arbitrary specificity of 2000:

you	3000
cost	6000
expensive	8000
NeuroStudio	8000

First, the selection of activators at compile-time is illustrated. The conjunctive IntermediateLevelCondition 1620 represents the conjunction of the Recall ?FactQuestion condition 1608 and the Heard “expensive” condition 1612. The matching condition 1612 has a specificity of 8000, while the recall condition 1608 has a specificity of only 2000. Therefore, condition 1612 is selected as the activator for IntermediateLevelCondition 1620, and IntermediateLevelCondition 1620 is assigned a specificity of 9000 (8000 + 2000 minus the “correlation factor” of 1000.) Similarly, the condition 1614 that tests for the word “cost” is selected over the recall condition 1610 as the activator for IntermediateLevelCondition 1622, and condition 1622 is assigned a specificity of 7000. As described above, disjunctive IntermediateLevelCondition 1624 is activated by both of its children, and assigned a compile-time specificity value of 7000, corresponding to the lowest specificity value among its children.

IntermediateLevelCondition 1624 (specificity 7000) is chosen over BaseLevelCondition 1616 (specificity 3000) as the activator for MatcherBlock 1600. BaseLevelCondition 1618 (specificity 8000) is chosen over BaseLevelCondition 1614 (specificity 6000) as the activator for IntermediateLevelCondition 1626, which is assigned a specificity of 13000 ($8000 + 6000 - 1000$). Finally, IntermediateLevelCondition 1626 (specificity 13000) is chosen over BaseLevelCondition 1610 (specificity 2000) as the activator for MatcherBlock 1602.

Now, suppose the matcher is given the input "Can you tell me the cost of NeuroStudio?" This input is classified as a ?DescriptionQuestion by the standard libraries, since it is really asking for information about the cost of NeuroStudio. In **Figure 16**, BaseLevelConditions 1610, 1614, and 1618 are active. Condition 1614 activates IntermediateLevelCondition 1622, while Condition 1618 activates IntermediateLevelCondition 1626. The matcher first checks the other children of IntermediateLevelCondition 1622 (in this case, only condition 1610) and finds that all are true, so condition 1622 is active, and has specificity 7000, using the same calculation as before. Similarly, the other children of IntermediateLevelCondition 1626 are checked, and condition 1626 is found to be true and given a specificity value of 13000. Condition 1622 then activates condition 1624, which is now guaranteed to be true since it is a disjunction. Since no other children of 1624 are true, Condition 1624 has a specificity of 7000 (it would have a specificity of 9000 if condition 1620 were true, as the run-time specificity of a disjunction is equal to the highest specificity of any true child.) Condition 1624 now activates MatcherBlock 1600. The matcher now checks the other children of MatcherBlock 1600 and finds them true. The block has a specificity of 9000 ($7000 + 3000 - 1000$) and is an active block. Similarly, condition 1626 activates MatcherBlock 1602. The other children of MatcherBlock 1602, in this case condition 1610, are true, so the block 1602 is active, and has a specificity of 14000 ($13000 + 2000 - 1000$). This block has a higher specificity than block 1600, so MatcherBlock 1602 is selected and condition #2 is chosen as the "best match" to the input.

Suppose instead that the matcher is given the input "Do you cost a lot?" According to the standard library of question types used in our examples, this is classified as a ?FactQuestion; therefore BaseLevelConditions 1608, 1614, and 1616 are active. Condition 1614 activates IntermediateLevelCondition 1622. The matcher then checks the other children of condition 1622 and finds that BaseLevelCondition 1610 is not true. Therefore IntermediateLevelCondition 1622 is not true and does not activate any other conditions. No other BaseLevelCondition activates any other condition, so neither matcher block is activated by this input.

VI. MECHANISM FOR PRONOUN REPLACEMENT

In certain cases, a method of direct pronoun replacement can be used to perform pronoun disambiguation more effectively than the automatic attention focus mechanism discussed above.

Topics in Gerbil can be assigned subjects using the **Subjects** keyword. These subjects are used to identify the set of related topics that are brought to the front of the attention focus stack whenever a topic is activated. In the present invention, additional information can be given for some subjects such that when such a subject is the focus of the current conversation, pronouns found in the user input can be replaced with appropriate words.

In the present embodiment, pronoun replacement information is assigned to subjects using the **SubjectInfo** keyword. **SubjectInfo** declarations appear at the top level of the GeRBiL script, along with **PatternList** declarations and other declarations.

Pronoun replacement is implemented by including a **CMapStringToString** structure **m_mssReplacements** in the **CUserRec** structure for each user, which contains the current mapping from pronouns to replacements. Initially, this mapping is empty. When a topic is executed that has a subject for which subject information has been declared as described above, each pronoun-replacement pair in the subject information declaration is added to the pronoun replacement map, replacing existing entries in the map when necessary. If a topic that is executed has more than one subject for which subject information has been declared, the map is updated in the reverse of the order in which the subjects are listed in the topic, insuring that the first subject listed has priority over other subjects listed.

Pronoun replacement is not done automatically, but occurs upon the execution of a **Compute** command with the function name **ReplacePronouns**, e.g.:

Remember ?WhatUserMeant is Compute ReplacePronouns of ?WhatUserSaid;

This pronoun replacement is implemented by scanning the argument(s) of **ReplacePronouns**, and for each word in each argument, seeing if it has an entry in the map structure **m_mssReplacements** for the current user. If so, the appropriate replacement word is substituted; otherwise the word is left unchanged.

A. EXAMPLES OF PRONOUN REPLACEMENT

Having described the present embodiment of the invention, the operation of the present invention will now be illustrated by describing the execution of a sample script, listed below.

TABLE 5 – EXAMPLES OF AUTOMATIC PRONOUN REPLACEMENT

SubjectInfo "Walter" is

Replace "he" with "Walter";
 SubjectInfo "Scott" is
 Replace "he" with "Scott";
 5 Priority Topic "Replace pronouns" is
 Always
 Remember ?WhatUserMeant is
 Compute ReplacePronouns of ?WhatUserSaid;
 Continue
 10 EndTopic

 Topic "Who is Walter?" is
 Subjects "Walter";
 IfHeard "who*Walter" Then
 15 Say "Walter is the president of Neuromedia";
 Done
 EndTopic

 Topic "Is Walter married?" is
 20 Subjects "Walter";
 IfHeard "Walter" and "married" Then
 Say "Walter is married to Maybellyne.";
 Done
 EndTopic
 25
 Topic "Other questions about Walter" is
 Subjects "Walter";
 IfHeard "Walter" Then
 Say "I don't know the answer to what you are asking about Walter.";
 30 Done
 EndTopic

 Topic "Who is Scott?" is
 Subjects "Scott";
 35 IfHeard "who*Scott" Then
 Say "Scott is the vice president of technology at Neuromedia.";
 Done
 EndTopic

 40 Topic "Other questions about Scott" is
 Subjects "Scott";
 IfHeard "Scott" Then
 Say "I don't know the answer to what you are asking about Scott.";
 Done
 45 EndTopic

For the purposes of illustration, assume that a site visitor engages in the following discussion with a site that has implemented the above script:

Transcript of interaction with BOT incorporating script example:

Console says: Who is Walter?

Robot says: Walter is the president of Neuromedia.

Console says: Is he married?

Robot says: Walter is married to Maybellyne.

Console says: Who is Scott?

Robot says: Scott is the vice president of technology at Neuromedia.

Console says: Is he married?

Robot says: I don't know the answer to what you are asking about Scott.

Execution of the script example:

When execution of the example begins, the replacement map `m_mssReplacements` in the user record for the visiting user is empty. When the first input, "Who is Walter?" is given to the BOT, the BOT first executes priority topic "Replace pronouns". Since the pronoun replacement map is empty, this topic has no effect, and the attribute `?WhatUserMeant` (used in `IfHeard`) is set to "Who is Walter?" The BOT now executes the standard topics, and selects the topic "Who is Walter?" as the best match to the input. (The topic "Other questions about Walter" is also activated, but is not considered as good a fit for the question. The other three topics do not match the input and so are not activated at all.) The output "Walter is the president of Neuromedia." is produced. Since the topic "Who is Walter" has subject "Walter", the subject information for "Walter" is added to the replacement map. In the above script, this causes a mapping from "he" to "Walter" to be added to the map. (In a more complex example, other words such as "his" and "him" might also be added.)

Next, the BOT is given the input "Is he married?". The BOT first executes the priority topic "Replace pronouns." The word "he" is found in the `m_mssReplacements` map and is thus replaced with "Walter", while the words "is" and "married" are not found in the map and are thus unchanged. Thus, the attribute `?WhatUserMeant` is assigned the value "Is Walter married?". The BOT selects the topic "Is Walter married?" as the best match to the input (again, "Other questions about Walter" is active but not selected, while the other three topics do not match the input) and the output "Walter is married to Maybellyne." is produced. Again, this topic has subject "Walter", so a mapping from "he" to "Walter" is added to the `m_mssReplacements` map. Since this mapping is already present, the replacement map is unchanged.

Next, the BOT is given the input "Who is Scott?" and the BOT executes the priority topic

“Replace pronouns” for this input. None of the words are found in the replacement map, so ?WhatUserMeant is simply assigned the value “Who is Scott?”. In this case, the topic “Who is Scott” is selected as the best match to the input, so the output “Scott is the vice president of technology at Neuromedia.” is produced. The topic “Who is Scott?” has subject value “Scott”, so the subject information for “Scott” is added to the replacement map. In this case, the mapping from “he” to “Scott” is added to the map, and overwrites the existing mapping from “he” to “Walter”.

Finally, the BOT is given the input “Is he married?”. The BOT executes the priority topic “Replace pronouns”, and this time replaces the word “he” with “Scott”, resulting in the value “Is Scott married?” for ?WhatUserMeant. This question activates only the topic “Other questions about Scott” so the output “I don’t know the answer to what you are asking about Scott.” is produced.

The above described example illustrates a behavior that would be more difficult to implement using only the best-fit matching and automatic focus of attention mechanisms. If best-fit matching is used and the topic “Is Walter married?” contains the pronoun “he” as well as Walter, the question “Is he married?” will activate the topic “Is Walter married?” even if the subject “Scott” is more focused. Using only the best-fit matching and automatic focus mechanisms, the only ways to write a script to respond to the above questions would be to either create a single topic each for Walter and Scott or include an “is Scott married” topic. Neither solution is easy to generalize for complicated BOT scripts.

VII. IMPLEMENTATION OF THE CONTEXT-TESTING MECHANISM

A. OVERVIEW

The condition **Focused** can be included in the BOT script by the BOT author, in the same way that other conditions such as **Heard** and **Recall** are used. Conceptually, the **Focused** condition is intended to be true whenever the category in which it is included is relevant to the current conversation. More specifically, the **Focused** condition can only be true in categories that have one or more subjects assigned using the **Subjects** keyword. The **Focused** condition is true if one or more of the subject keywords associated with the category are “active subjects”.

The set of active subjects is the set of subject keywords associated with the most recent input that was processed and resulted in at least one topic associated with a subject being focused, either automatically or through a FOCUS command, as well as subject keywords focused using a FOCUS SUBJECTS command. All subjects associated with each topic that was focused are included in this set of active subjects. Thus, a topic that does not contain any subject keywords

does not change the set of active subjects. For instance, a default response such as "I don't know" might not have any associated subjects, and therefore would not change the subject of the current conversation. Thus, the user can ask a question after such a default response and assume that the bot will "remember" what the context of the question was.

In the present implementation of the mechanism, the user record structure for each user conversing with the BOT contains a mapping `m_mspActiveSubjects` that contains an entry for each currently active subject. These entries actually map to NULL pointers; the map structure is used for efficient lookup. When one or more categories are activated or focused using a FOCUS command, and at least one of these categories is associated with one or more subject keywords, or a FOCUS SUBJECTS command is executed, this map is cleared and a new entry is made for each such keyword. When all of the categories that are activated by an input are not associated with any keywords, the map `m_mspActiveSubjects` remains unchanged. It will be appreciated by one skilled in the art that this set of active subjects might instead be stored as an ordered list or in an array, and that the scope of the present invention should not be limited to any particular such implementation. Similarly, other reasonable protocols might be used to update this map, including clearing the map when a category without subject keywords is activated, or keeping the subject keywords from two or more previous category activations rather than only the most recent, and that the scope of the present invention should not be limited to the particular protocol described above.

In the compiled BOT object created from the script, each **Focused** condition is represented as follows:

```
class CFocusCondition: public CCondition
{
public:
    CCategory*    m_pcCategory;
    BOOL          m_bIsOptional;
    BOOL          Evaluate(CGRBLToolDoc* Context);
    void          ProduceFailureWarning(
                    CGRBLToolDoc* Context);
    LPCTSTR       Name() {return "Focused";};
    CActionType   WhatItIs() {return ActIfFocused;};
    BOOL          ContainsStar() {return FALSE;};
    void          MakeOptional();
    CFocusCondition() {
        m_bIsOptional = FALSE;
        m_pcCategory = NULL;
    };
};
```

One element of the **Focused** condition object is the pointer `m_pcCategory` to the category that contains it. In order to evaluate the condition, the system examines each of the elements of the array of subject names found in the category `m_pcCategory`. Each of these names is looked up in the map `m_mspActiveSubjects` for the current user. If one or more of the names is found in the map, the **Focused** condition is true; otherwise, it is false. It will be appreciated that other methods of evaluating **Focused** conditions would be apparent to one skilled in the art, such as directly maintaining a list of all categories in which a **Focused** condition is true, and that the scope of the present invention should not be limited to the particular implementation disclosed above. It will likewise be appreciated that other forms of conditions apart from Boolean conditions, such as fuzzy logic conditions, modal logic conditions, are well known to those skilled in the art and are hereby subsumed by the scope of the present invention.

In the present implementation of the mechanism, each **Focused** condition is included in the best-fit matching selection structure as a run-time `BaseLevelCondition`. The value of these conditions can be computed using a run-time matching object, but these values are not computed unless the block containing the **Focused** keyword is activated by some other condition. Since there may be many topics containing **Focused** conditions, it is more efficient to compute the values of each such condition only when needed. Each **Focused** `BaseLevelCondition` includes an array of the subject names associated with the category containing the condition; this array can be used at run-time to evaluate the condition when needed. It will be appreciated that other methods of incorporating **Focused** conditions into such a best-fit matching structure would be apparent to one skilled in the art and that the scope of the present invention should not be limited to the particular implementation disclosed above.

Like other run-time `BaseLevelConditions`, **Focused** conditions in the best-fit matching selection structure are given a compile-time specificity of 0, so that they are never chosen as activators for categories. At run-time, a **Focused** condition is assigned a specificity of 100 times the number of subjects in common between the category containing it and the currently active subjects. **Focused** conditions are deliberately assigned a low specificity value so that they do not interfere with more specific answers to user questions. It will be appreciated that other ways of assigning specificity values to **Focused** conditions could also be used. In particular, if an ordered active subjects list is maintained, the specificity of a **Focused** condition could depend on the position in which the subjects in the category containing appear in the active subjects list.

In certain cases, particularly when there is more than one subject in the active subjects list, the use of **Focused** conditions may result in two or more answers with identical specificity. In this case, the focus of attention stack will function as disclosed above to select an appropriate

response. Suppose the list of active subjects includes both "Walter" and "Scott", and the question "Who is he?" is included in both a "Who is Walter?" and a "Who is Scott?" topic using the **Focused** keyword in each case. In this case, both possible answers will have the same specificity, so the topic that is nearer the front of the focus of attention will give the answer. The focus of attention stack itself is unaffected by the **Focused** mechanism.

B. EXAMPLE OF FOCUSED MECHANISM

Having described the architecture of a present embodiment, the operation of the present invention will now be illustrated by describing the execution of an example script. Table 6 illustrates a BOT script in the present scripting language, using the **Focused** condition in several places. This simple script answers several questions about the company Neuromedia and the scripting language Gerbil.

TABLE 6 -- AN EXAMPLE SCRIPT PROGRAM

```

Topic "What is Neuromedia" is
Subjects "NEUROMEDIA";
    If (Heard "what*Neuromedia") or
      (Focused and heard "what*it", "what*that")
    Then
      Example "What is Neuromedia?";
      Say "Neuromedia is a small corporation that sells bot authoring software";
    Done
EndTopic

Topic "Where is Neuromedia" is
Subjects "NEUROMEDIA";
    If (Heard "where*Neuromedia") or
      (Focused and Heard "where")
    Then
      Example "Where is Neuromedia";
      Say "Neuromedia is located in San Francisco";
    Done
EndTopic

Topic "What is Gerbil" is
Subjects "GERBIL";
    If (Heard "what*Gerbil") or
      (Focused and Heard "what*it")
    Then
      Example "What is Gerbil";

```

```

    Say "Gerbil is a bot-scripting language";
    Done
EndTopic

5
Topic "Is Gerbil easy?" is
Subjects "GERBIL";
    If (Heard "is*Gerbil*easy") or
        (Focused and (Heard "is*it*easy"))
10
    Then
        Example "Is Gerbil easy to use?";
        Say "Yes, Gerbil is very easy to use.";
    Done
15
EndTopic

Topic "Give me an example of Gerbil" is
Subjects "GERBIL";
    If (Heard "example" and "Gerbil") or
20
    (Focused and heard "example")
    Then
        Example "Give me an example of Gerbil";
        Say "Here's a sample of a Gerbil script:",
25
        "    Topic \"Hello World\" is",
        "    If Heard \"hello\" Then",
        "        Say \"Hi there!\"",
        "    Done",
        "    EndTopic";
    Done
30
EndTopic

Default Topic "I don't know" is
    Always
35
    Say "I don't know what you mean.";
    Done
EndTopic

=====

For purposes of illustration, assume that a site visitor engages in the following discussion
40 with a site that has implemented the script shown in Table 6.

```

Transcript of Interaction with Robot incorporating script in Table 6:

```

User Says: What is Neuromedia?
Robot Says: Neuromedia is a small corporation that sells bot authoring software
45
User Says: where?
Robot Says: Neuromedia is located in San Francisco

```

User Says: what is Gerbil?

Robot Says: Gerbil is a bot-scripting language

User Says: where?

5 Robot Says: I don't know what you mean.

User Says: is it easy?

Robot Says: Yes, Gerbil is very easy to use.

10 User Says: Give me an example

Robot Says: Here's a sample of a Gerbil script:

Robot Says: Topic "Hello World" is

Robot Says: If Heard "hello" Then

Robot Says: Say "Hi there!"

15 Robot Says: Done

Robot Says: EndTopic

User Says: Where is Neuromedia

Robot Says: Neuromedia is located in San Francisco

20

User Says: What is it?

Robot Says: Neuromedia is a small corporation that sells bot authoring software.

User Says: Give me an example

25 Robot Says: I don't know what you mean.

Execution of the Sample Script in Table 6:

The script shown in Table 6 contains 6 categories: two discussing the subject

30 "Neuromedia", three discussing the subject "Gerbil", and one default topic, with no subject, that simply gives the response "I don't know what you mean." to any question that isn't answered by any of the other topics. It is assumed here that the same user is talking to the robot throughout this example.

35 > User Says: What is Neuromedia?

> Robot Says: Neuromedia is a small corporation that sells bot authoring software

Initially, the active subject map for this user is empty. When the user ask the question

"What is Neuromedia?", the only topic that is activated is the "What is Neuromedia?" topic.

40 (This input is handled exactly as it would have been if the script had been written without the use of the present invention; the topic "What is Neuromedia?" is the only topic that would be activated regardless of the contents of the active subject map.) Therefore, the robot gives the response "Neuromedia is a small corporation that sells bot authoring software." Since the topic

“What is Neuromedia?” was activated, the subject “Neuromedia” is added to the active subject map for the user. (If the topic included more than one subject, all of these subjects would have been added to the active subject map.)

- 5 > User Says: where?
 > Robot Says: Neuromedia is located in San Francisco

The user now asks the question “where?”. This question contains the word “where”, so the best-fit selection mechanism activates the second clause of the condition in the topic “Where is Neuromedia?” (i.e. **Focused and Heard “where”**.) It now tests the **Focused** condition by testing whether any subject of the “Where is Neuromedia?” topic is contained in the active subject map for the current user. In this case, the subject “Neuromedia” is included in the active subject map, so the **Focused** condition is true, and the category “Where is Neuromedia?” is selected. When the category is actually executed, a similar process is used to evaluate the **Focused** condition and the robot says “Neuromedia is located in San Francisco.” The topic “Where is Neuromedia” contains the subject “Neuromedia” so the active subject map still contains only the single subject “Neuromedia”.

Note that this response would be very difficult to produced without the use of a **Focused** condition. Pronoun replacement clearly could not produce this response, as the input did not contain any pronouns. The focus of attention stack could be used to produce this response, but might also produce the same response if the question “where?” is asked in a different context. As shall be illustrated, the script in Table 6 does not suffer from this problem.

- 25 > User Says: what is Gerbil?
 > Robot Says: Gerbil is a bot-scripting language

The next question, “What is Gerbil?”, matches only the topic “What is Gerbil”, independent of the set of active subjects. Therefore, the robot produces the response “Gerbil is a bot-scripting language”. Since the topic “What is Gerbil” contains at least one subject, the active subject map for the current user is cleared out, and the subject “Gerbil” is added to the map.

- > User Says: where?
 > Robot Says: I don't know what you mean.

35 The user now repeats the question “where?”, but in a different context. Again, the question activates the second clause of the topic “Where is Neuromedia?” within the best-fit

selection mechanism. It now tests the **Focused** condition and discovers that none of the subjects of the topic "Where is Neuromedia?" are found in the active subject map for the current user. Therefore, the **Focused** condition is false, and the topic "Where is Neuromedia?" is not activated. No other topics are activated by the question "where?", so the robot executes the default topic "I don't know" and produces the response "I don't know what you mean." Since the only topic activated was the "I don't know" topic, and this topic does not contain any subjects, the active subject map for the current user is unchanged (i.e. it still contains the single subject "Gerbil".) Note that without the use of the **Focused** condition, it would be difficult to avoid having the robot produce the answer "Neuromedia is located in San Francisco" in response to this question.

> User Says: is it easy?
 > Robot Says: Yes, Gerbil is very easy to use.

The user now asks the question "Is it easy?". The only condition that is activated by this input is the condition (**Focused and (Heard "is*it*easy")**) in the topic "Is Gerbil easy?". The best-fit matching mechanism now evaluates the **Focused** condition and determines that it is true, since the topic "Is Gerbil easy?" is associated with the subject "Gerbil", which is found in the active subject map for the current user. Therefore, the response "Yes, Gerbil is very easy to use." is given. Since the topic "Is Gerbil easy?" contains only the subject "Gerbil", the active subject map remains unchanged.

> User Says: Give me an example
 > Robot Says: Here's a sample of a Gerbil script:
 > Robot Says: Topic "Hello World" is
 > Robot Says: If Heard "hello" Then
 > Robot Says: Say "Hi there!"
 > Robot Says: Done
 > Robot Says: EndTopic

The user now says "Give me an example". This question activates the condition (**Focused and Heard "example"**) in the topic "Give me an example of Gerbil". Again, the **Focused** condition is tested and found to be true, since the topic is associated with the subject "Gerbil", which is found in the active subject map for the current user. The response is given, and the active subject map remains unchanged.

> User Says: Where is Neuromedia?
 > Robot Says: Neuromedia is located in San Francisco

The user now asks the question “Where is Neuromedia?” This question matches the condition (**Heard “where*Neuromedia”**) in the “Where is Neuromedia” topic, so the answer “Neuromedia is located in San Francisco” is given. The active subject map for the current user is now cleared, and the subject “Neuromedia”, from the topic “Where is Neuromedia”, is added to the active subject map.

> User Says: What is it?

> Robot Says: Neuromedia is a small corporation that sells bot authoring software.

The user now asks the question “What is it?” This question activates two different conditions, the condition (**Focused and Heard “what*it”**) in the topic “What is Neuromedia”, and the identical condition in the topic “What is Gerbil”. For the condition within the topic “What is Neuromedia”, the **Focused** condition is tested and found to be true, as the subject “Neuromedia” is in the active subject map. For the condition within the topic “What is Gerbil”, the **Focused** condition is found to be false, since the subject “Gerbil” is not in the active subject map. Therefore, the “What is Neuromedia” topic is the only topic activated, and the answer “Neuromedia is a small corporation that sells bot authoring software” is given. The topic contains the subject “Neuromedia”, so the active subject map remains the same.

> User Says: Give me an example

> Robot Says: I don't know what you mean.

Finally, the user repeats the input “Give me an example”. Again, this input activates the condition (**Focused and Heard “example”**) within the topic “Give me an example of Gerbil”. This time, the **Focused** condition is tested and found not to be true, since the subject “Gerbil” is not in the active subject map. Therefore, the default topic “I don't know” is executed, and the bot says “I don't know what you mean.”

VII. IMPLEMENTATION OF AUTOMATIC BOT VERIFICATION

A. OVERVIEW

In the present embodiment of the invention, the BOT author enables verification by including verification statements, referred to as “Example statements” in the BOT script. These Example statements are ignored during ordinary execution of the BOT. However, if the BOT author wishes to test the performance of the BOT, these statements are used as test cases for the

verification mechanism. Further detail on the structure and use of these Example statements is given below.

Figure 18 gives an overview of one way in which the example verification mechanism might be used. The BOT author 122, when creating the BOT script, includes example statements in the BOT script 108. When the BOT script is compiled by the compiler 107, the resultant BOT object 106 contains the examples. When the BOT author 122 desires to test the BOT, they first select the set of examples 1800 which are to be tested, consisting of all or some of the entire set of examples in the BOT script. These examples are tested by the verification mechanism 1802, using the BOT object 106, as will be discussed in greater detail below. The result of this testing is a report file 1804 detailing the errors that were found in the BOT object. The BOT author 122 can then use this report file 1804 to make corrections 1806 to the BOT script 108. The changed BOT script can then be recompiled by the compiler 107 and the process can be repeated as often as desired.

In the present BOT scripting language, Example statements are included within the body of a topic. Each Example statement represents a particular input that an end user might give to the BOT. Currently, these Example statements are limited to textual input. It will be appreciated that Example statements can easily be extended to include non-textual input as well, and that the present invention should not be limited to textual Example statements.

An Example statement is normally included inside an IF block that also contains one or more **Say** or **Do** statements. The inclusion of an Example statement in such a block indicates that, in response to each of the input strings given in the Example statement, the author intends the BOT to respond by executing the **Say** and/or **Do** statements contained in the block. Thus, the Example statement also serves as a helpful reminder to the BOT author of the input that the **Say** or **Do** statement was intended to respond to.

In practical usage, the BOT author will often want to include multiple Example inputs for a single block. In order to make the script more readable, these additional Example inputs can be placed outside the original topic through the use of the **OtherExamples** statement. An **OtherExamples** statement associates the original Example input with additional inputs that are intended to have the same response.

In addition, there may be inputs which are intended to produce different answers when asked in different circumstances. For instance, the question "Who is he?" may produce several different responses depending on the prior conversation. Such inputs can be tested by including them in an **OtherExamples** statement and including the **WhenFocused** keyword. The meaning of such examples is that they are intended to produce the appropriate answer if asked when the

topic associated with the original Example is already the current focus of the conversation. (For instance, if the Focus of Attention mechanism described earlier is being used, the appropriate answer should be provided given that the topic is at the front of the Focus of Attention.) If the topic containing the original Example is not the current focus of the conversation, these

5 Examples may produce any answer. As an example, consider the input question "Who is he?" This question should produce an answer describing Walter if Walter is the focus of the current conversation, while it may reasonably produce other answers if somebody else is the focus of the conversation. Such inputs need to be tested in a context-sensitive fashion, so that they do not produce false error messages.

10 After the BOT script is compiled, Examples are included within the Executable Program Object 106 in three places. First, they are included along with other actions in the body of the If..Then block where they are found. Second, a list of all the Examples found within the script is included in the CProgram structure, for easy access during verification. These Examples are split up into three types: standard examples, used for ordinary inputs; initial examples, used for
15 initialization; and sequence examples, used to test sequential behaviors. These three types of examples are discussed further below. Finally, a mapping is included in the CProgram structure that contains all the **OtherExamples**, if any, associated with each string given as the argument of an **Example** statement, and information as to whether they are context-dependent examples or not.

20 In the current implementation, OtherExamples can only be specified for a fixed string, although the present invention should not be limited to the use of fixed strings when specifying variations of an Example.

An example of a Topic in our scripting language that contains several Example statements is given below in Table 7.

25

=====

TABLE 7 – A SCRIPT PROGRAM CONTAINING EXAMPLE STATEMENTS

Topic "Walter" is
 30 IfHeard "who*is" and ("Walter", "he") Then
 Example "Who is Walter?";
 Say "Walter is the president of Neuromedia";
 Done
 IfHeard "Walter" Then
 Example "Do you know who Walter is?";
 Say "You asked some question about Walter...";
 35 Done

EndTopic

=====

5 B. THE EXECUTION OF EXAMPLE STATEMENTS

 In order to verify the performance of a BOT, the BOT author first includes Example statements in the BOT script, as described in the previous section, while writing and modifying the BOT script. In the course of ordinary execution of the BOT script, these Example statements
10 are completely ignored. In order to verify the performance of the BOT, the BOT author selects a verification command within the BOT development environment and then selects a set of Examples to verify. For instance, the author may select all the Examples within a particular file, or all the examples in the entire BOT script. These Examples are verified, as described below, and a text file report is produced detailing the errors that were found and, optionally, the correct
15 responses that were given. It will be appreciated that there are other plausible interfaces to the Example verification mechanism, such as an interactive presentation of the results or the creation of a database containing the results, and that the present invention should not be limited to the production of a text file report of the results of verification.

 As stated above, Examples are included within the BOT Executable Program Object in
20 three places: within the body of If..Then blocks, in one of three lists of examples within the CProgram structure, according to example type, and in a mapping of OtherExamples in the CProgram structure.

Figure 19 shows the process of example verification in a BOT. Example verification begins at 1900. At 1902, the user selects the set of examples to be verified, and may select a
25 number of options for the format of the output from the verification mechanism; for instance, the user may elect not to have each input and response output to the console window, as would be done during ordinary operation of the BOT. The verification mechanism then (at 1904) does any necessary initialization of the run-time environment 106 for the BOT, for instance to correctly set the output options chosen by the user at 1902. In the present implementation, this step
30 insures that information from the run is stored in an Example report file, makes sure that output from the BOT is not passed on to any external connections the BOT may have with web pages, chat sites, etc., and sets appropriate flags in the execution program so that it will know to generate appropriate error messages when an Example produces incorrect output. It would be obvious to one skilled in the art that other methods of interface for a verification mechanism are

possible, and that the scope of the present invention should not be limited to any particular interface for selecting the set of examples to be verified (including the possibility of automatically selecting all examples) or to any particular set of output options or interface for selecting such options.

5 The verification mechanism now iterates 1906, 1908 through the Initial Examples as described in the following section. It now saves the state of the user at 1909 so that all examples can be verified in an equivalent user state. It then iterates 1910, 1912, 1914, 1915 through the Sequence Examples as described in the following section. It now iterates 1916, 1918, 1920, 1921 over each of the standard examples. Since the results of the mechanism are intended to be
10 independent of the order in which the Examples are tested, they can be tested in arbitrary order, although the current implementation tests them in the order in which they appear in the original scripts. In order to insure that the results are order-independent, the execution mechanism stores the state of the user at 1909 and restores this state at 1915, 1921 after each Example is tested, thus insuring that all Examples are tested in the same user state. It would be obvious to one
15 skilled in the art that other ways of ordering the Examples to be tested are possible and that the scope of the present invention should not be limited to any particular method of ordering. Likewise, it would be obvious to one skilled in the art that Examples could be tested without restoring the user state before testing each Example.

 Once an Example is selected from the list, it is tested 1918 to see whether it is one of the
20 Examples that the user wishes to test – for instance, by examining the name of the file containing the Example. If it is, the Example is verified. The process 1920 of verifying each Example is described in more detail below. Finally, the user state stored at 1909 is restored at 1921, allowing further examples to be tested in the same state. Once iteration is complete, the verification mechanism resets 1922 the run-time environment 106 to its original settings, and
25 terminates at 1924.

Figure 20 shows the process of verification of a single Example in the BOT, and corresponds to operation 1920 in **Figure 19**. Execution starts at 2000. The verification mechanism first sets (at 2002) flags in all the **If** statements containing the example. As each command in the present scripting language is inside one or more **If** blocks, these flags may be
30 used to produce diagnostic error messages if the Example does not produce the correct output. It then evaluates at 2003 the expressions (e.g. ordinarily fixed strings) that are arguments to the Example statement, in the standard way in which expressions are evaluated in the scripting language, producing a list of strings. It then iterates 2004 over each element of this list of strings. Each iteration starts at 2006 by running the script on the string, in the same way that the

script would be run on an external input (although potentially not producing the same external output, due to the flags set at 1904.) Since a BOT script must be run with respect to a particular user, a “test user” record is created, or retrieved if one has already been created, in order to pass to CProgram::Run. This user record is used to store the Attention Focus List, the User Attribute
5 Memory, and other properties of the hypothetical “conversation” that is taking place between the verification mechanism and the BOT.

The program handles the input exactly as it would have handled the same input from an external user. However, by default, Examples are verified in a “find all equivalent matches mode” that is different from the mode in which inputs are ordinarily processed. This mode can
10 be turned off by the BOT author in the input selection process at 1902. In this mode, once a **Done** has been reached in a standard category, the program checks to see if there are any other categories that are active and have the same specificity as the category containing the **Done**. These categories are then executed, despite the **Done** that has already been reached. If only one category was selected, it indicates that the answer would have been given regardless of the
15 context in which the input occurred. However, if more than one category was selected during this execution, it must be true that each of these categories would have been executed first if it had been the category at the front of the focus of attention. Therefore, each such category is a possible answer that might be given in response to the input in some context, and the set of such categories represents all such possible answers.

Once the argument has been run, the verification mechanism produces (2008) an error message, if needed, describing any errors that were found. Any time that a **Say** or **Do** statement is executed in response to an Example input, it is considered to have produced a response to the input (this includes any **Say** or **Do** statements that are executed in categories executed due to
20 “find all equivalent matches mode”). There are three possible cases that can occur in response to an Example input:
25

1. If the execution process has executed the body of the **If** block that directly contains the Example statement, and no responses were given other than those found in that **If** block, then the Example is considered to have produced the correct answer. If the BOT author has chosen to
30 have messages printed in response to correct answers, a message will be output to the Example report, otherwise the verification of that Example is complete.

2. If the execution process has executed the body of the **If** block but has also given responses other than the **If** block, a warning message is produced in the report file. This message

can include, for instance, information about the other responses that were given and which topic was responsible for them, and the order in which the correct and incorrect responses were given.

3. If the execution process did not execute the body of the **If** block, the Example statement clearly did not produce the correct answer. This case can be further divided into the seven cases shown in Table 8 below. In each of these cases, useful diagnostic information can be included in the Example verification report regarding the cause of the error. First, if the category containing the Example was executed but the body of the **If** block was not, there are two possible cases. Either some **If** condition surrounding the example failed, or a **Done** or **NextTopic** was executed earlier in the category. If one of the **If** conditions failed, the flags that were set at 2002 in the conditions surrounding the Example will cause error messages to be printed out to the report file during the execution explaining why the **If** statement failed. If no such error message was printed out, the verification mechanism can deduce that the reason for the failure must have been that a **Done** or **NextTopic** was executed earlier in the category. Next, if the Example is in a Priority category that never got executed, a **Done** must have been executed in some earlier Priority category. Similarly, if the Example is in a Default category that never got executed, a **Done** must have been executed by some Priority, standard, or Sequence category, or an earlier Default category. Next, if the Example is in a Sequence category that never got executed, the Sequence category was clearly never called. Finally, if the Example is in a standard category that never got executed, there are two possible cases. Such a category was either not activated by the best-fit mechanism because some **If** statement within the block surrounding the example was not true, or was activated (i.e. all conditions in some block in the category were true) but was not selected. If the category was not active, the category containing the Example is executed anyway at 2010. Since the category was not active, there must be at least one **If** condition surrounding the Example that is not true, and since a flag was set for each of these **If** conditions, an error message will be generated when the category is executed explaining why the Example did not produce the correct response. If on the other hand the category was active, the verification mechanism at 2010 executes all remaining categories that are at least as specific as the one containing the Example, producing information as to all the other responses that might be given in preference to the desired response for the Example.

TABLE 8: POSSIBLE CAUSES FOR LACK OF CORRECT RESPONSE TO AN
EXAMPLE

Category containing Example executed?	Type of category containing Example	Cause
Yes	Any	If condition failed
	Any	Done earlier in category
No	Priority	Done in earlier Priority category
	Default	Done in any earlier category
	Sequence	Sequence category not called
	Standard	If condition failed
	Standard	Higher priority category selected

Once an argument of the Example statement has been executed, the verification mechanism first stores the user state at 2012 so that it can be used for testing Examples

5 WhenFocused. It now iterates 2014, 2015, 2016, 2018, 2020 over any other inputs that are included in **OtherExamples** statements as being equivalent to the original argument.

OtherExamples are included in the Program Executable structure as a map from input strings to lists of other examples. Therefore, in order to test the other examples associated with an input, the verification mechanism simply needs to look up this input in the Other Example map and test
10 each of the examples found there. In order to remove any dependence of the results on the order in which the OtherExamples are tested, the user state is restored 2015 to the state in which the user was before the original example was run at 2006. This "original state" was saved earlier, at 1909. The examples found in normal **OtherExamples** statements are then tested as given above, by 2016 running the script, 2018 generating error messages, and 2020 executing any other
15 categories. The verification mechanism now iterates 2022, 2024, 2026, 2028, 2030 over any examples found in **OtherExamples** statements with the **WhenFocused** keyword. The mechanism currently assumes that the execution of the original argument at 2006 is enough to make the subject the focus of attention, and thus restores at 2024 the user state that was stored at 2012 (otherwise other topics may inadvertently become the focus of attention in the course of
20 execution of intermediate examples.) The "WhenFocused" example is now run on the script at 2026, generates error messages at 2028, and executes other categories as needed at 2030.

However, when the example is run at 2026, it is run without the "find all equivalent matches mode" described above. This will test if the input produces the correct response when that category is the primary focus of attention -- which is what the user wishes to know -- while not
25 testing what response will be given if the category is not the primary focus of attention. Thus, the use of the WhenFocused keyword allows context-sensitive examples to be tested without generating spurious error messages warning that these context-sensitive examples might produce other responses.

As an alternative embodiment, at step 2024, the mechanism could instead of restoring a saved user state, directly modify the state of the focus of attention mechanism to make the category containing the original example the focus of attention. This implementation would also allow the "WhenFocused" example to be tested in a state in which said category is the primary focus of attention.

After all of the Other Examples of an argument of an Example statement have been verified, the verification mechanism returns to 2004 and continues with the next argument. Once it has verified all of the arguments, it resets at 2032 the flags that were set in the **If** conditions surrounding the Example at 2002, and returns to the caller at 2034 to continue with the next Example.

C. EXTENSIONS TO THE EXAMPLE MECHANISM

Verifying examples is similar to having a new user log in and type in all the Examples. Ordinary Examples are tested in arbitrary order, which can create a problem if a BOT has some kind of initialization routine such as a routine that asks for the user's name when they first log in. For instance, the first Example tested may produce the response "Hi, what is your name?" rather than a response to the example.

The Example verification mechanism described in the previous section can be extended to cover initialization routines by including a set of examples in the initialization routine that will be tested before any other Examples and will be tested regardless of the subset of Examples chosen for verification by the BOT author.

In the present implementation of the scripting language, these Examples are indicated by using the **InitialExample** statement. **InitialExample** statements are included in the initialization routines. Each InitialExample is given a non-negative integer (its **index**), immediately following the keyword **InitialExample**, that determines the order in which the InitialExamples are executed, and has exactly one argument. Otherwise, InitialExamples are equivalent to regular Examples. It is an error for more than one InitialExample to have the same index, or for an InitialExample to have more than one input string.

When the BOT script is compiled, all the InitialExamples found in the script are sorted according to index and stored in an array in the ProgramExecutable structure. At the beginning of each run of the Example verification routine, all of these InitialExamples are tested in order, at 1906 and 1908. All of the InitialExamples are run regardless of which examples were chosen for verification by the BOT author. The execution of the InitialExamples is done in the same method as shown in **Figure 20**, although in the present implementation, OtherExamples are not

used with InitialExamples, and the InitialExamples are never run in “find all matches mode.” (Although since InitialExamples are normally included only in Priority categories, the “find all matches mode” would not make a difference in the result.) Note that the user state is saved 1909 after the execution of the initial examples, since the initial examples prepare the state of the user for the execution of the standard examples.

The Example verification mechanism described in the previous section is primarily useful for testing inputs in which the order of inputs is irrelevant. The mechanism can be easily extended to verify “sequences” of interaction as well. “Sequence” in this case refers to any series of interactions that are order-dependent, such as interactions in which the BOT asks a question of the user and processes one or more possible responses. Sequences are generally more difficult to test, because all of the possible branches of the interaction must be tested.

For instance, consider the following code fragment:

```

15      Topic “Monitor diagnosis” is
          IfHeard “screen*black” Then
              Say “Is the monitor turned on?”;
              WaitForResponse;
              IfHeard “yes” Then
                  Say “Is it plugged in?”;
20              WaitForResponse;
                  IfHeard “yes” Then
                      SwitchTo “Problem diagnosis”;
                      Done
                  Otherwise Always
25                      Say “Try plugging it in”;
                      Done
              Done
              Otherwise Always
30                      Say “The on switch is located on the front”;
                      Done
          Done
      EndTopic

```

In order to test this fragment, the initial question (“My screen is blank”) has to be repeated at least 3 times, once for each possible line of answers (and probably more if the “problem diagnosis” sequence has multiple lines of answers.) This testing can be handled automatically by an extension of the Example mechanism.

In order to test a sequence in the present implementation of the invention, the BOT author first determines the input that initiates the interaction, in this case “My screen is blank”. For the purpose of testing, we consider the “interaction” to be the set of all possible interactions that can

arise directly from that initial input. In the present implementation of the invention, this initial input is assigned an integer as an "index", but the scope of the invention should not be limited to any particular indexing scheme for the examples that make up a sequence.

Next, the BOT author considers all the possible user responses that can follow the initial input. In this case, the robot responds to the initial input by asking "is it turned on?", to which there are two possible categories of user responses (discounting responses such as "I don't know" and irrelevant responses). Each response is assigned a letter or word that gets appended to the end of the index for that response. For instance, the response "yes" might be given the index "170.yes" or "170.y" while the response "no" might be given the response "170.no" or "170.n". Note that the letter or word used in the index is not used directly in testing the examples; only the actual text in the Example statement is used. Again, the scope of the present invention should not be limited to any particular naming scheme for these indices.

This process is continued for any user responses that might follow such user responses. In this case, the user response "yes" is followed by the robot response "Is it plugged in?", which would have two possible responses once again. As before, each of these is assigned a letter or word that is added to the existing index. For instance, the response "no" might be assigned the index "170.yes.no" while the response "yes" might be assigned the index "170.yes.yes". Note that once again, the index "170.yes.yes" is not directly used in testing; it merely indicates that the response "Yes" is intended as a follow-on to the interaction that led to the Example statement for "170.yes". Examples of further interactions within the "Problem diagnosis" topic would have indexes beginning with "170.yes.yes...".

The Sequence does not contain any further interaction after the user response "no" and robot response "The on switch is located on the front", so there are no further examples continuing from "170.no".

In the current implementation of the scripting language, the index for each response is included in the **Example** statement immediately following the keyword **Example**, much like the **InitialExample** statement. Unlike **InitialExample**, however, testing of sequences is done with the standard **Example** keyword. Thus, the syntax of the examples described above is:

Example 170 "My screen is blank";
 Example 170.yes "Yes";
 Example 170.no "No";
 Example 170.yes.no "No";
 Example 170.yes.yes "Yes";

It is an error to have more than one Example with the same index, or to have an index nX.Y (where n is an integer, Y is a string, and X is zero or more strings with a period before each string) for which there is no Example with an index nX.

When the BOT script is compiled, the sequence Examples are collected into groups according to the initial number in the index of each. We define the “original Example” in each group as the Example for which the index consists only of an integer. (As indicated in the above paragraph, there must be such an example for each group.) These original Examples are sorted and stored in an array in the ProgramExecutable structure. Each sequence Example, including the original Examples for each group, includes a mapping that includes each possible Example that can immediately follow the Example, according to the definitions above. **Figure 22** illustrates the Sequence example structure that is generated from the code fragment above once the Example statements described above are added. The original example 2200 for the group contains the example text “My screen is blank” and pointers to the two follow-up examples 2202 and 2204. The first follow-up 2202 has index “170.yes” and input text “yes”, and contains pointers to two follow-ups 2206 and 2208. The other follow-up 2204 has index “170.no” and input text “no”, and does not contain any follow-up links. Similarly, the follow-ups 2206 and 2208 to the example “170.yes” do not have any follow-up links. The execution of the example set shown in **Figure 22** is discussed below.

Figure 19 shows the execution of the sequence Examples occurring during the iteration 1910, 1912, 1914, 1915 immediately following the InitialExamples. The iteration 1910, 1912, 1914, 1915 occurs over only the original sequence example in each group; the remainder of the sequence examples are tested during the execution process 1914. The original example is first tested 1912 to see whether it is one of the Examples that the user wishes to test, as was done for the standard examples at 1918. If it is selected, it is executed 1914 by a special execution process described below, and the user state is reset 1915 so that further examples may be tested in the same state. It would be obvious to one skilled in the art that sequence Examples could equivalently be tested along with or after the standard Examples, and that the scope of the present invention should not be limited to testing sequences at any particular time.

Figure 21 illustrates the execution of a single group of Sequence Examples, corresponding to the execution process at 1914. Execution starts at 2100, and begins by executing 2102 the original example for the group, using the execution process shown in **Figure 20**. If test 2104 determines that the example was given an incorrect answer, further testing of the sequence would not produce meaningful results (since further examples in the sequence make the assumption that earlier examples produced the correct result), and execution is terminated at

2106. If instead a correct answer was given, the state of the test user is saved at 2108, so that each follow-up to the Example can be executed in the same context. (Note that if there are multiple arguments and/or OtherExamples defined for the Example, they are tested during the execution 2102 to insure that they produce the same result.) The verification mechanism now
 5 iterates 2110, 2112, 2114 over each possible follow-up to the example. Each follow-up is executed at 2112 with a recursive call to the same process shown in **Figure 21**, since the follow-up is itself a Sequence Example and may have its own follow-ups. After execution of the follow-up is complete, the user state (stored in 2108) is restored at 2114, and iteration continues at 2110. Once iteration is complete, control returns to the caller at 2116.

10 Thus, the result of the Example verification is equivalent to the result that would be generated if all possible sequences of Examples were tested, but without the possibility of having to test a number of inputs that is exponential in the number of Examples given. The detailed methods used in the compilation and execution of sequences of Examples would be well known to one skilled in the art.

15 In the present implementation of the mechanism, the initial statement in each sequence interaction is tested in "Find All Matches" mode; all other statements are not since they are assumed to follow the initial statement.

20 D. EXAMPLES OF BOT VERIFICATION USING EXAMPLE STATEMENTS

We will now discuss several instances of the example verification mechanism in operation. The BOT script shown below includes several Example statements. For explanatory purposes, the following BOT script contains intentional errors in order to illustrate the testing mechanisms of the present invention.

```

25 Topic "Walter" is
    IfHeard "who*is" and ("Walter", "he") Then
        Example "Who is Walter?";
        Say "Walter is the president of Neuromedia";
30 Done
    IfHeard "Walter" Then
        Example "Do you know who Walter is?";
        Say "You asked some question about Walter...";
    Done
35 EndTopic

Topic "Where Walter works" is
    IfHeard "work#" Then

```

Example "Who is Walter working for?";
Say "Walter works for Neuromedia";

Done

EndTopic

5

OtherExamples of "Who is Walter?" are
"Who is Dr. Tackett?";
OtherExamples of "Who is Walter?" WhenFocused are
"Who is he?";

10

Suppose that the BOT author chooses to verify all the Example statements in the above BOT script. For this description, we will assume that the BOT author has chosen to print out information about both correct and incorrect examples. The entire output of the verification mechanism is shown below in Table 9.

15

There are no InitialExamples or Sequence Examples in the script, so after initialization, the verification mechanism begins with the first standard example, "Who is Walter?". Following the flowchart in **Figure 20**, the **IfHeard** condition preceding the Example is marked (at 2002), the arguments of the Example are evaluated 2003 (in this case, evaluation is trivial) and the script is run (at 2006) on the input "Who is Walter". In this case, the script produces the correct response, and a message indicating that the response is correct is output (at 2008.) No additional categories need to be executed, so the user state is saved (at 2012).

20

The verification mechanism now iterates over all OtherExamples of the input, and begins with the input "Who is Dr. Tackett?". The original user state is first restored 2015, in this case producing no relevant changes. The script is next run on the example (at 2016). In this case, no category is selected as the input does not match any set of conditions in the BOT script. Since the category containing the Example ("Walter") is a standard category, and since no higher priority category was selected, the verification mechanism can deduce that an **if** condition must have failed. Therefore, as presented in Table 8 above, the verification mechanism runs the category "Walter". Since the **IfHeard** statement was marked (at 2002), when it is executed it outputs a message saying that the pattern "Who is Dr. Tackett?" did not match any of the patterns.

30

All standard OtherExamples of the input "Who is Walter?" have now been tested, and the verification mechanism now iterates over all OtherExamples WhenFocused of the input, beginning with the input "Who is he?". It first restores the user state at 2024 (it was saved at 2012), insuring that the topic "Walter" is the focus of attention (although it does not matter in this case.) It then runs the script on the input at 2026, and receives the correct answer, so a message is output at 2028 indicating that the response is correct.

35

The verification mechanism has now completed all OtherExamples for the input “Who is Walter?” and returns to the iteration 2004. Since there are no other arguments of this Example statement, iteration terminates, the **IfHeard** condition is un-marked at 2032, and the verification mechanism moves on to the next example.

5 The next example is “Do you know who Walter is?”, found in the next block in the topic “Walter”. The block is marked (2002) and the example is run through the BOT program (2006). It produces the response “Walter is the president of Neuromedia”, which is not the response that corresponds to the Example statement (although in this case it is a reasonable response.) Since the **Say** statement that was executed is outside the block, an error messages is output during
10 execution, indicating that an incorrect response was given. Finally, once execution is complete, the verification mechanism determines that the block containing the Example was not executed, so an error message is printed (at 2008.) Since the category was executed, and the block was not, execution must have terminated earlier in the category, and an appropriate messages is output. There are no OtherExamples corresponding to the example “Do you know who Walter
15 is?”, so execution switches to the next Example.

Finally, the example “Who is Walter working for?” is tested. In this case, the input matches blocks in both topics “Walter” and “Where Walter works”. The BOT execution mechanism selects the topic “Walter” as being a better fit to the question. Therefore, the answer “Walter is the president of Neuromedia” is given. Since this answer is not in the block
20 containing the Example, an error message is output during execution. Once execution is complete, the verification mechanism determines that the category containing the Example was not executed. Since in this case, the category is a standard category and some other standard category was executed, the other category must have been selected as a more appropriate answer. As described earlier, the verification mechanism executes the category containing the Example at
25 2010, and provides appropriate information about both the example category and the category that was executed first instead.

There are no further examples to be tested, so summary information is output and the verification operation is complete. A transcript of the output from the verification mechanism in this case is shown in Table 9.

TABLE 9 – OUTPUT FROM TESTING OF EXAMPLE SCRIPT

Information for Example "Who is Walter?":

>>>Selecting: Category 'Walter'

Correct answer given:

Robot Says: Walter is the president of Neuromedia

5

Information for Example "Who is Dr. Tackett?":

ERROR: Example "Who is Dr. Tackett?" in Topic 'Walter' (in Example.g line 3) was not hit because:

Input 'Who is Dr. Tackett?' did not match any pattern

10

Information for Example "Who is he?" WhenFocused:

Correct answer given:

Robot Says: Walter is the president of Neuromedia

15

Information for Example "Do you know who Walter is?":

>>>Selecting: Category 'Walter'

Incorrect answer given:

Robot Says: Walter is the president of Neuromedia

20

ERROR: Example "Do you know who Walter is?" in Topic 'Walter' (in Example.g line 7) was not hit because:

An If block earlier in Topic 'Walter' was executed instead - see above

Information for Example "Who is Walter working for?":

>>>Selecting: Category 'Walter'

25

Incorrect answer given:

Robot Says: Walter is the president of Neuromedia

ERROR: Example "Who is Walter working for?" in Topic 'Where Walter works' (in Example.g line 14) was not hit because:

Best answer had specificity value 3753

30

>>>Selecting: Category 'Where Walter works' (Specificity 3169)

Correct answer given:

Robot Says: Walter works for Neuromedia

Summary: 5 Examples were tested.

35

2 were correct, 0 gave the correct answer plus others, 3 did not give the correct answer.

=====

The following topic illustrates the use of InitialExamples. A name capture topic such as this might be found in many BOT scripts. If InitialExamples are not given for this topic, the first example tested by the verification mechanism would produce the response "Hi, what's your name?", the second would produce the response "What is your email address?", and so on. Once the three InitialExamples given below are added to the topic, these examples are given to the BOT before any other examples, regardless of which examples were selected for verification by the user.

Priority Topic "Name Capture" is

Always

```
// "hi" here functions as a dummy first input;
InitialExample 1 "hi";
Say "Hi, what's your name?";
WaitForResponse;
// "Example Tester" would be given as the response.
InitialExample 2 "Example Tester";
Remember ?Name is ?WhatUserSaid;
Say "What is your email address?";
WaitForResponse;
// the address below would be given as the response
InitialExample 3 "nobody@neurostudios.com";
Remember ?Email is ?WhatUserSaid;
Say "Please ask me some questions";
Suppress this;
// other examples can now be tested without interference
// from Topic "Name Capture".
```

Done

EndTopic

The following topic illustrates the use of sequence examples, and is based on the discussion of sequence examples in the previous section.

Topic "Monitor diagnosis" is

IfHeard "screen*black" Then

Example 170 "My screen is blank";

Say "Is the monitor turned on?";

WaitForResponse;

IfHeard "yes" Then

Example 170.yes "Yes";

Say "Is it plugged in?";

WaitForResponse;

IfHeard "yes" Then

Example 170.yes.yes "Yes";

SwitchTo "Problem diagnosis";

Done

Otherwise Always

Example 170.yes.no "No";

Say "Try plugging it in";

Done

Done

Otherwise Always

Example 170.no "No";

Say "The on switch is located on the front";

Done

Done

EndTopic

Figure 22 illustrates the SequenceExample data structure created from the above script. The initial example in the sequence is Example 170, "My screen is blank". This example is stored as sequence example 2200 and is kept in an array in the ProgramExecutable structure, indexed by the index 170. The sequence example 2200 has pointers to all possible follow-ups, sequence example 2202 (the example 170.yes above) and sequence example 2204 (the example 170.no above.) There are no follow-ups to example 170.no, but example 170.yes has two follow-ups, sequence example 2206 (the example 170.yes.yes above) and sequence example 2208 (the example 170.yes.no above).

Execution of the sequence examples follows the flowchart given in **Figure 21**. After starting at 2100, the first example 2200, with text "My screen is blank", is executed at 2102, exactly as standard examples are run. The example run terminates at the **WaitForResponse** immediately following the question "Is the monitor turned on?". Assuming that the answer given is correct, the user state following the example run (including the information associated with the **WaitForResponse**) is saved at 2108. The sequence example 2200 has two follow-ups, so the execution routine is recursively called on follow-up sequence example 2202. Execution on example 2202 begins at 2100 and runs the example at 2102. Execution resumes at the **WaitForResponse**, so the text value "yes" for the example 2202 leads to the response "Is it plugged in?". This is the correct answer, so the user state (which is currently at the **WaitForResponse** following "Is it plugged in?") is saved at 2108, and execution continues with the follow-ups to sequence example 2202. First, the execution mechanism is recursively called on sequence example 2206. Execution resumes after the **WaitForResponse** after "Is it plugged in?", so the text value "yes" for the example 2206 produces the correct response by switching to "Problem diagnosis". The user state at this point is saved, but since sequence example 2206 contains no follow-up examples, execution terminates at 2116 and execution returns to sequence example 2202. Execution for this example now restores (at 2114) the user state that was saved at 2108 (the state that was at the **WaitForResponse** following "Is it plugged in"). Iteration continues with follow-up sequence example 2208, which is equivalent to the execution of sequence example 2206. Once execution of sequence example 2208 is complete and control returns to the execution of sequence example 2202, the user state is restored once again. This time there are no further follow-up examples, so execution on sequence example 2202 terminates and control returns to the execution on sequence example 2200. The user state that was saved after the sequence example 2200 was originally executed (i.e. the **WaitForResponse** after the

question “Is the monitor turned on”) is restored, and iteration continues with follow-up sequence example 2204. The text “no” for this example produces the correct response “The on switch is located on the front”. There are no follow-ups to sequence example 2204, so control returns to the execution for sequence example 2200. There are no further follow-ups to sequence example 2200, so execution for this sequence example is complete.

The execution described above effectively tested the following three complete interactions, although saving and restoration of user states were used to simulate the repetition of the initial statement and the “yes” response to the first question. Note that if the initial statement or the “yes” response had produced an incorrect response, testing of the follow-ups would not have been done, as the follow-ups could not have been tested in their correct context.

User says: My screen is blank
 Robot says: Is the monitor turned on?
 User says: Yes
 Robot says: Is it plugged in?
 User says: Yes

User says: My screen is blank
 Robot says: Is the monitor turned on?
 User says: Yes
 Robot says: Is it plugged in?
 User says: No

User says: My screen is blank
 Robot says: Is the monitor turned on?
 User says: No
 Robot says: The on switch is located on the front

C. BACK-END FEATURE LEARNING FROM EXAMPLE INPUTS

Figure 23 shows the architecture for automatic learning of features in the context of the preferred embodiment of the present invention. The learning mechanism is controlled by the learning and validation manager 2300, which is a user interface to present input options and summary of progress to the interactive user. It will be appreciated that mechanisms for visual presentation are well known by those skilled in the art.

The validation manager 2302 is a combination of components previously described which processes an historical sample of inputs using code created by the learning process in

order to estimate and predict the quality of the automatically generated code. The Learning Tool 2301 embodies one mechanism for learning Gerbil code from a set of example inputs previously categorized by a human operator. Its major components include: feature extractor 2303, as shown in **Figure 24**, which reduces an input sample to a set of abstract representations called features; learner 2304, as shown in **Figure 25**, which, given an input sample set where each input element has a label, selects a subset of the extracted features so that each feature is contained in many but not all of the input samples, creating a partition of the input sample set into those which either do or do not contain the feature, so that the label of an input sample can be predicted with some confidence based upon which members of the subset of extracted features it contains; report module 2305, as shown in **Figure 26**, for generating statistical and diagnostic outputs associated with the automated learning process; code generator 2306, as shown in **Figure 27**, which converts the subset of features selected by the learning process, along with a representation of the associated labels, into Gerbil source code, via substitution of learned features into the appropriate fields of a preexisting template. The mechanisms for creation of an instance of source code by substitution of specific fields into a template are appreciated by a practitioner of the art.

Figure 28 shows the process of feature extraction 2303 in greater detail, following the processing of one raw data sample. The flowchart **Figure 24** shows the algorithm for this process. A raw data sample 2800 can be but is not limited to a string of text, representing an action or communication from a user, which may have a label associated with it. Without loss of generality, this configuration will be used to describe the feature extraction 2303 and labeling process.

The feature extraction process **Figure 24** begins at 2400, with the most basic form of input is from a user or a historical database 2402 of prior user inputs, containing only alphanumeric characters, spaces, and punctuation. The end of the input is typically delimited by a special character such as <TAB>, immediately followed by a label designating the category, character, or properties of the sample. Typically this label is assigned by a human "teacher," and has a one-to-one correspondence with the action that the teacher wishes for the robot to take in response to input of this category. The inputs are processed until reaching end-of-file 2422, at which point there is a control transfer 2424 to the learner module flowchart **Figure 25**.

The extraction phase 2801, 2802 for features consists of converting the input 2800, 2420 to a standard form so that its components can be more readily stored and analyzed. Typically, this might consist of converting all characters to a uniform case 2404, and reduction of the text

input to a vector of word “tokens” with all spaces and nonessential punctuation removed 2406, 2408.

Feature encoding 2803, 2804 consists of converting textual tokens to a set of symbolic codes, which denote not only the occurrence of the token 2410, but special properties it may have, such as being the leading, trailing, or only token in the input 2412, or co-occurrence with one or more tokens also contained in the current input 2414. For example, there is a unique feature assigned to the word-token “who” occurring at the beginning of a sentence. There may be a second unique feature assigned to the word “who” occurring in a sentence regardless of position, and both features will then be generated in response to one input sentence which has the word “who” at the beginning. Other entries may be formed from compound features comprising multiple words, such as “who is” or “who knows”, which may occur within the same sentence frequently.

Database insertion 2805, 2410, 2412, 2414, consists of entering the encoded representations of the features into a feature dictionary, and creating statistics associated with the features. Each unique feature is isomorphic to a unique code that represents it in the feature dictionary 2418. The feature dictionary 2418 is organized so that there is only one entry for each symbolic code representing feature f , and so that the occurrences and the label(s) associated with them are tabulated in the dictionary entry. In the entry for each label l there is a counter $p(l, f)$ 2416 of the co-occurrences of label l with feature f . For the example above, if the label of the sample is “LabelType1,” then the two unique codes generated will each have an entry in the feature dictionary, and each entry will have a counter associated with LabelType1 which is incremented.

After all data in the sample set have been processed, the feature dictionary 2418 contains one entry for each type of feature that was encountered in the sample set. Each entry has a count 2416 of the number of times the corresponding feature occurred in the input set. Furthermore, this count is broken down further by label for each feature, so that a unique correspondence between features and labels can be identified.

Figure 25 shows the flowchart for the learner 2304, while **Figure 29** shows the associated processing of the feature dictionary 2418 in tabular form 2900 with features represented in the rows by letters A...H (nominally there are thousands, rather than 8), and Label histogram for each feature represented by columns, numbered 1...4 (the number of labels must be at least 2, and can easily range in the hundreds).

The algorithm for learning starts at 2500, iterating 2506, 2512, over elements of Feature Dictionary 2418. Counter elements $p(f, l)$ 2416 for the feature dictionary 2418 table 2900 entries

are just the estimated joint frequency of occurrence for each Feature-Label pair. When each entry is divided by the sum of all entries, it yields the estimated joint probability of occurrence, designated $p(\text{Feature}, \text{Label})$. In practice, it is an advantage to not divide through by this sum, as shown in the **Figure 25** flowchart 2502, 2504, 2508, 2510, since a large number of

unnecessary and computationally expensive division operations can be avoided. Summing the columns in each row X produces an estimate of the marginal frequency with which each Feature(X) will occur, denoted $p(\text{Feature} = X)$, or just $p(X)$. Likewise, summing row values in each column N yields an estimate of the marginal probability that Label(N) will occur, denoted $p(\text{Label}=N)$, or $p(N)$. In practice, this is achieved by summing rows and columns of $p(f,l)$ to produce the sums SumF(L) 2502, and SumL(F) 2504 $p(f,l)$.

Figure 29 shows the way of computing conditional probabilities of labels on features 2901. Dividing each row element $p(\text{Feature}, \text{Label})$ of the joint probability table by the corresponding $p(\text{Feature}=X)$ yields the table of estimated conditional probability $p(\text{Label}|\text{Feature})$ that Label= N is the correct label for an input sample given that Feature= X was detected. The actual algorithm **Figure 25** divides each element $p(f,l)$ 2416 by the sum SumL(F) 2504 to yield the conditional probability $p(l|F)$ 2510. Note that $p(\text{Label}|\text{Feature})$ is the identical result $p(l|F)$ obtained by the algorithm, since there exists a sum S so that $p(\text{Feature}, \text{Label}) = p(f,l)/S$ for each Feature= f and Label= l , and $p(\text{Label}=N) = \text{SumL}(N)/S$.

Figure 29 also shows the way of computing conditional probabilities of features on labels 2902. Dividing each column element of $p(\text{Feature}, \text{Label})$ by the corresponding $p(\text{Label}=N)$ yields another table of estimated conditional probability $p(\text{Feature}|\text{Label})$ that Feature= X is present in an input sample given that the sample is actually of type Label= N . The algorithm **Figure 25** performs this computation by dividing $p(f,L)$ 2416 by SumF(L) 2502 to produce $p(f|L)$ 2508.

After conditional probabilities 2508 and 2510 are computed, the algorithm Flowchart **Figure 25** iterates again 2514, 2522, over the Feature/Label pairs, to yield reliable predictive features 2526. **Figure 30** depicts the data flow of this process. Two thresholds are chosen: the "row" threshold, **Thresh₁** 3000, 2516, and "column" threshold, **Thresh₂** 3001, 2518.

Thresh₁ 3000, 2516, is set to insure that a feature is strongly predictive of a label: a Feature= X can be considered for use only if it can predict that a sample has Label= N with probability greater than or equal to **Thresh₁** 3000, 2516. In other words, if and only if $p(\text{Label}|\text{Feature}) > \text{Thresh}_1$ 2510, 3000, 2516. Note that 1.0 is greater than or equal to $p(\text{Label}|\text{Feature})$ 2510, which is greater than or equal to 0.0, and **Thresh₁** 3000, 2516 is normally in the range greater than 0.9. Since all elements $p(\text{Label}|\text{Feature}=X)$ 2510 sum to 1.0

for a given row, then any value greater than 0.5 insures that at most one entry in a row can be considered.

Thresh₂ 3001, 2518 is set to insure that a feature occurs frequently enough to be of use. A **Feature=X** can be considered for use only if it occurs with probability greater than or equal to **Thresh₂** 3001, 2518 when an input sample with **Label=N** occurs. In other words, if and only if $p(\text{Feature}|\text{Label}) > \text{Thresh}_2$ 2902, 3001, 2518. In general, no feature will reliably occur in all cases of **Label=N**, and so it is desirable for several diverse instances **Feature=X_i** to surpass the threshold in order to provide adequate "coverage" so that for any given input some feature can be detected which confidently predicts **Label=N**.

Therefore, the detected feature set which is subsequently used to form Gerbil™ source code is a set of features that meet the criteria:

$$p(\text{Label}|\text{Feature}=\text{X}) > \text{Thresh}_1 \text{ AND } p(\text{Feature}=\text{X}|\text{Label}) > \text{Thresh}_2$$

Where for a given **Feature=X** there is only one label satisfying both clauses of the disjunction. This union of conditions is denoted by the intersection of shaded areas 3002 in **Figure 30**. All features which meet the above threshold criteria are added to the Predictive Features Database 2526. After all the features have been processed 2512, control is transferred 2524 to the Report Module 2305, flowchart **Figure 26**.

Figure 26 shows how Report Module 2305 iterates over the contents of Predictive Feature Database 2526 to produce an output file 2612. Upon starting 2600, the algorithm iterates 2602, 2608, over the Predictive Feature Database 2526. Each feature f_i in the Predictive Feature Database 2526 is looked up 2604 in the Feature Dictionary 2418, and the Label, $P(f_i|l)$, and $P(l|f_i)$ for that entry are written 2606 to the Predictive Feature Report File 2612. Upon completion, control is transferred 2610 to Code Generator module 2306 flowchart **Figure 27**.

Table 10 shows a typical output file from Report Module 2305 for feature extractor 2303 and learner 2304 run against a database of approximately 2800 labeled input sentences, with **Thresh₁**=0.9 and **Thresh₂**=0.1. The first column shows a unique serial number generated for each feature. The second column shows a representation of the corresponding feature, according to the following legend:

<i>Feature</i>	<i>Meaning</i>
word1	word occurring anywhere in input line
<word1	word occurring at start of input line
>word1	word occurring at end of input line
^word1	only word in input line

	word1 word2	word2 occurring directly after word1 in input line
	word1*word2	word2 occurring after word1, with 0 or more occurrences of other words between
5	word1 AND word2	word1 and word2 occurring in input line in any order

Inspection of the table shows that the process can be optimized to eliminate redundancy.

10 In column 3, the categories chosen in this example represent 10 types of questions that
are being distinguished from each other. Note that many of the questions in this database were
concerning the filing of taxes, and so there is a particular bias in the category of “**WhatIf**”
questions towards words like “**file**” and “**late**.” Columns 4 and 5 show the conditional
probabilities being tested.

15 In the same way that report module 2305 generates a report, the Code Generator module
2306 generates corresponding Gerbil source code 2712. Upon starting 2700, the algorithm
iterates 2702, 2708, over the Predictive Feature Database 2526. Each feature f_i in the Predictive
Feature Database 2526 is looked up 2704 in the Feature Dictionary 2418, and the feature f_i for
that entry is written as an activation pattern, and an associative memory element isomorphic to
the Label is remembered as the action corresponding to the activation, and the resulting Gerbil
20 code is written 2706 to the Gerbil File 2712. An example of this is shown in **Listing** below.
Upon completion, the execution of Learning Tool 2301 is complete 2710.

The results shown in Table 10 can be used to directly generate Gerbil source code, since
the features translate directly into Gerbil pattern matching expressions described earlier, and the
labels can immediately take the form of Gerbil associative memory patterns. The probabilities
25 are not directly used by the Gerbil code, but are included here only as an illustration of typical
probability values for this algorithm.

Listing shows excerpts from a typical output file from Code Generator module 2306
using results from feature extractor 2303 and learner 2304 run against a database of
approximately 2800 labeled input sentences, with **Thresh₁**=0.9 and **Thresh₂**=0.1 (the same data
30 shown in Table 10).

Listing shows one clause in an automatically generated Gerbil topic, representing the
first and last entries in the table (intermediate examples are omitted for brevity):

Listing

Topic "Autogenerated Question Classifier" is
 If ?WhatUserSaid Matches "#can*#I*" Then
 Remember ?CategoryLabel is "CanYouI";
 Done
 ...
 If ?WhatUserSaid Matches "#who #are*" Then
 Remember ?CategoryLabel is "Who";
 Done
 EndTopic

Table 10

Serial #ID	Feature	Label	p(F L)	p(L F)
3650	<can*i	canyoui	54.69%	100.00%
3651	<can i	canyoui	51.56%	100.00%
3711	<can*get	canyoui	18.75%	92.31%
3802	<can*you	canyoui	15.63%	100.00%
52	<may	canyoui	15.63%	90.91%
3803	<can you	canyoui	12.50%	100.00%
5720	<what*is	describe	24.68%	91.77%
5721	<what is	describe	22.93%	94.71%
5723	<what*are	describe	13.85%	97.54%
5724	<what are	describe	11.41%	100.00%
1858	<is	fact	18.87%	100.00%
18	<are	fact	10.25%	98.57%
18603	<what about	followup	65.52%	100.00%
18674	<how about	followup	18.97%	100.00%
2898	<where	location	67.24%	100.00%
18877	<where*i	location	34.48%	100.00%
18854	<where*the	location	31.03%	100.00%
18964	<where*can	location	28.74%	100.00%
18965	<where can	location	27.01%	100.00%
18879	<where*get	location	15.52%	100.00%
18863	<where*is	location	13.79%	100.00%
18864	<where is	location	13.22%	100.00%
19339	<where*find	location	10.92%	100.00%
18874	<where*do	location	10.34%	100.00%
9438	<how*i	method	47.52%	93.20%
10598	<how do	method	41.58%	96.55%
20215	<how can	method	16.34%	100.00%
8654	<how*can	method	16.34%	97.06%
11066	<how*get	method	14.85%	96.77%
3289	<why	reason	71.70%	99.13%
21999	<why*the	reason	15.72%	100.00%
22141	<why*you	reason	14.47%	100.00%
21990	<why*do	reason	13.84%	100.00%
21991	<why do	reason	12.58%	100.00%
22024	<why*is	reason	10.06%	100.00%
3485	<should	should	68.97%	100.00%
23419	<should i	should	41.38%	100.00%
23418	<should*i	should	41.38%	100.00%
3518	<when	time	64.29%	100.00%
23658	<when*i	time	21.43%	100.00%
23743	<when*the	time	14.29%	100.00%

23905 <what if	whatif	57.14%	100.00%
23992 happens AND if	whatif	25.00%	100.00%
23991 happens if	whatif	25.00%	100.00%
23990 happens*if	whatif	25.00%	100.00%
23943 <what*>date	whatif	21.43%	100.00%
23937 <what*file	whatif	21.43%	100.00%
23939 <what*later	whatif	21.43%	100.00%
23941 <what*than	whatif	21.43%	100.00%
3544 >date	whatif	21.43%	100.00%
23946 file AND if	whatif	21.43%	100.00%
23962 file AND later	whatif	21.43%	100.00%
23964 file AND than	whatif	21.43%	100.00%
23966 file AND that	whatif	21.43%	100.00%
23967 file*>date	whatif	21.43%	100.00%
23961 file*later	whatif	21.43%	100.00%
23963 file*than	whatif	21.43%	100.00%
23965 file*that	whatif	21.43%	100.00%
23955 i AND later	whatif	21.43%	100.00%
23956 i*>date	whatif	21.43%	100.00%
23954 i*later	whatif	21.43%	100.00%
23949 if AND later	whatif	21.43%	100.00%
23951 if AND than	whatif	21.43%	100.00%
23952 if*>date	whatif	21.43%	100.00%
23945 if*file	whatif	21.43%	100.00%
23948 if*later	whatif	21.43%	100.00%
23950 if*than	whatif	21.43%	100.00%
3543 later	whatif	21.43%	100.00%
23978 later AND than	whatif	21.43%	100.00%
23980 later AND that	whatif	21.43%	100.00%
23977 later than	whatif	21.43%	100.00%
23981 later*>date	whatif	21.43%	100.00%
23976 later*than	whatif	21.43%	100.00%
23979 later*that	whatif	21.43%	100.00%
23984 than that	whatif	21.43%	100.00%
23985 than*>date	whatif	21.43%	100.00%
23983 than*that	whatif	21.43%	100.00%
23988 that >date	whatif	21.43%	100.00%
23987 that*>date	whatif	21.43%	100.00%
991 <who	who	75.76%	97.09%
24246 <who*is	who	17.42%	100.00%
24361 <who is	who	15.91%	100.00%
24254 <who*the	who	14.39%	100.00%
24138 <who*are	who	12.88%	100.00%
24139 <who are	who	12.12%	100.00%

Claims

1. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a set of predefined categories is activated, a method for focusing the attention of the automated interface program, the steps of said method comprising:

defining a list of categories activatable by said program;

maintaining an attention focus list, said focus list comprising an ordered list of categories;

executing actions associated with categories that are activated;

re-ordering said attention focus list during the course of interaction, said re-ordering being dependent upon which categories were activated.

2. The method for focusing the attention of said automated interface program as recited in claim 1, wherein the step of defining a list of categories further comprises:

defining a set of priority categories such that said priority categories are executed before all other categories.

3. The method for focusing the attention of said automated interface program as recited in claim 1, wherein the step of defining a list of categories further comprises:

defining a set of default categories such that said default categories are executed after all other categories.

4. The method for focusing the attention of said automated interface program as recited in claim 1, wherein the step of defining a list of categories further comprises:

defining a set of default categories such that said default categories are executed if no other categories produce output to the user.

5. The method for focusing the attention of said automated interface program as recited in claim 1, wherein the step of defining a list of categories further comprises:

defining a set of sequence categories such that said sequence categories are executed only when called by other categories.

6. The method for focusing the attention of said automated interface program as recited in claim 1, wherein the step of defining a list of categories further comprises:

defining a set of sequence categories such that said sequence categories produce a series of sequential interactions with the user.

5 7. The method for focusing the attention of said automated interface program as recited in claim 1, wherein said attention focus list further comprises a subset of standard categories.

10 8. The method for focusing the attention of said automated interface program as recited in claim 1, wherein the step of executing actions associated with categories that are activated further comprises:
 identifying a set of categories comprising priority, standard, sequence, and default categories activated by said input.

15 9. The method for focusing the attention of said automated interface program as recited in claim 8, wherein the step of identifying a set of categories activated by said input further comprises:
 testing conditions in categories such that said category is activated if said condition is satisfied by said input.

20 10. The method for selecting categories and executing actions associated with said categories as recited in claim 8, wherein the step of identifying a set of categories activated by said input further comprises:

- (i) testing conditions in said priority categories such that a priority category is activated if said condition is satisfied by said input;
- 25 (ii) identifying previously activated sequence categories that have yet to complete their actions according to a previous input;
- (iii) testing conditions in said standard categories such that a standard category is activated if said condition is satisfied by said input;
- (iv) testing conditions in said default categories such that a default category is
- 30 activated if said condition is satisfied by said input.

11. The method for focusing the attention of said automated interface program as recited in claim 1, wherein the step of executing actions associated with categories that are activated further comprises:

executing actions according to the order in which said categories appear in said attention focus list.

12. The method for focusing the attention of said automated interface program as recited in claim 11, wherein the step of executing actions according to said order appearing in said attention focus list further comprises:

executing actions according to the order in which standard categories appear in said attention focus list.

13. The method for focusing the attention of said automated interface program as recited in claim 1, wherein the step of re-ordering said attention focus list further comprises:

re-ordering said attention focus list automatically according to categories activated by user input.

14. The method for focusing the attention of said automated interface program as recited in claim 13, wherein the step of re-ordering said attention focus list automatically further comprises:

moving categories activated by said user input to the front of said attention focus list.

15. The method for focusing the attention of said automated interface program as recited in claim 13, wherein the step of re-ordering said attention focus list automatically further comprises:

- i) associating subjects with categories;
- ii) moving a first category that shares an associated subject with a second category activated by said user input to the front of said attention focus list.

16. The method for focusing the attention of said automated interface program as recited in claim 1, wherein the step of re-ordering said attention focus list further comprises:

re-ordering said attention focus list according to explicit commands within categories activated by user input.

17. The method for focusing the attention of said automated interface program as recited in claim 16, wherein the step of re-ordering said attention focus list according to explicit commands further comprises:

5 moving a first category that is explicitly mentioned in a focus command within a second activated category to the front of said attention focus list.

18. The method for focusing the attention of said automated interface program as recited in claim 16, wherein the step of re-ordering said attention focus list according to explicit commands further comprises:

10 i) associating subjects with categories;
ii) moving a first category that is associated with a subject explicitly mentioned in a focus subjects command within a second activated category to the front of said attention focus list.

15 19. The method for focusing the attention of said automated interface program as recited in claim 16, wherein the step of re-ordering said attention focus list according to explicit commands further comprises:

preventing a first category that is explicitly mentioned in a suppress command within a second activated category from being activated.

20

20. The method for focusing the attention of said automated interface program as recited in claim 19, wherein the step of preventing a first category from being activated further comprises:

25 enabling the activation of said first category that has been prevented from being activated, said first category explicitly mentioned in a recover command within a second activated category.

21. The method for focusing the attention of said automated interface program as recited in claim 16, wherein the step of re-ordering said attention focus list according to explicit commands further comprises:

30 preventing a category that is explicitly mentioned in an initial suppress command from being activated, said initial suppress command being executed upon compilation.

22. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a set of predefined categories is activated, a mechanism for focusing the attention of the automated interface program, said mechanism comprising:

- 5 a means for defining a list of categories activatable by said program;
- a means for maintaining an attention focus list, said focus list comprising an ordered list of categories;
- a means for executing actions associated with categories that are activated;
- a means for re-ordering said attention focus list during the course of interaction,
- 10 said re-ordering being dependent upon which categories were activated.

23. A machine-readable medium comprising an encoding, said encoding comprising a mechanism for focusing the attention of an automated interface program, said mechanism comprising:

- 15 a means for defining a list of categories activatable by said program;
- a means for maintaining an attention focus list, said focus list comprising an ordered list of categories;
- a means for executing actions associated with categories that are activated;
- a means for re-ordering said attention focus list during the course of interaction,
- 20 said re-ordering being dependent upon which categories were activated.

24. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a set of predefined categories is activated, a method for selecting categories and executing actions associated with said categories, the steps of said method comprising:

- 25 (a) defining a list of categories activatable by said program;
- (b) for an input,
 - (i) identifying a first set of categories activated by said input;
 - (ii) selecting a second set of categories from said first set of activated categories
 - 30 based upon a metric, said metric computed based on the conditions located within each said activated category;
 - (iii) executing actions associated with said second set of categories.

25. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of defining a list of categories further comprises:

5 defining a set of priority categories such that said priority categories are executed before all other categories.

26. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of defining a list of categories further comprises:

10 defining a set of default categories such that said default categories are executed after all other categories.

27. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of defining a list of categories further comprises:

15 defining a set of default categories such that said default categories are executed if no other categories produce output to the user.

28. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of defining a list of categories further comprises:

20 defining a set of sequence categories such that said sequence categories are executed only when called by other categories.

29. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of defining a list of categories further comprises:

25 defining a set of sequence categories such that said sequence categories produce a series of sequential interactions with the user.

30 30. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of identifying a first set of categories activated by said input further comprises:

identifying a first set comprising priority, standard, sequence, and default categories activated by said input.

31. The method for selecting categories and executing actions associated with said categories as recited in claim 30, wherein the step of identifying a first set of categories activated by said input further comprises:

testing conditions in categories such that said category is activated if said condition is satisfied by said input.

32. The method for selecting categories and executing actions associated with said categories as recited in claim 30, wherein the step of identifying a first set of categories activated by said input further comprises:

(i) testing conditions in said priority categories such that a priority category is activated if said condition is satisfied by said input;

(ii) identifying previously activated sequence categories that have yet to complete their actions according to a previous input;

(iii) testing conditions in said standard categories such that a standard category is activated if said condition is satisfied by said input;

(iv) testing conditions in said default categories such that a default category is activated if said condition is satisfied by said input.

33. The methods of selecting categories and executing actions associated with said categories as recited in claim 31, wherein the step of testing conditions further comprises:

(i) constructing a graph in which the nodes represent predetermined states of user input and the edges represent user input transitions between said nodes;

(ii) associating a set of final nodes with conditions within categories, wherein one of said conditions is satisfied if transitions upon a given user input place the state of user input at the associated final node.

34. The method for selecting categories and executing actions associated with said categories as recited in claim 33, wherein the step of selecting a second set of categories from said first set of activated categories based upon a metric, said metric computed based on the conditions located within each said activated category, further comprises:

said metric based on the length of the path in the graph traversed to a final node of a condition.

35. The method for selecting categories and executing actions associated with said categories as recited in claim 34, wherein the step of selecting a set of categories from said first set of activated categories based upon a metric further comprises:

selecting categories among said first set of categories for which the path length is maximized.

36. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of selecting a second set of categories from said first set of activated categories based upon a metric, said metric computed based on the conditions located within each said activated category, further comprises:

said metric based on the frequency with which said conditions are expected to be true.

37. The method for selecting categories and executing actions associated with said categories as recited in claim 36, wherein said metric further comprises:

(i) defining conditions to be patterns of user input;

(ii) estimating the frequency of conditions occurring in user input.

38. The method for selecting categories and executing actions associated with said categories as recited in claim 37, wherein the step of estimating the frequency of conditions occurring in user input further comprises:

estimating the frequency of conditions occurring in example user inputs included in category scripts.

39. The method for selecting categories and executing actions associated with said categories as recited in claim 36, wherein the step of selecting a set of categories from said first set of activated categories based upon a metric further comprises:

selecting categories among said first set of categories for which the conditions associated with the categories are least likely to be true.

40. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of selecting a set of categories from said first set of activated categories based upon a metric further comprises:

(i) associating metrics with the specificity of inputs supplied by a user;

5 (ii) selecting said categories from said first set of categories whose metrics are associated with the most specific input currently supplied by said user.

41. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of selecting a set of categories from said first
10 set of activated categories based upon a metric further comprises:

said metric being a numeric value computed for each activated category, the value being based upon the frequency of matched words, partial words, and symbols found in the current input with words, partial words, and symbols found in the conditional clauses located within the category.

15 42. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of selecting a set of categories from said first set of activated categories based upon a metric further comprises:

selecting said category having more matched words, partial words, and symbols
20 in the conditional clause located within the category.

43. The method for selecting categories and executing actions associated with said categories as recited in claim 24, wherein the step of selecting a set of categories from said first set of activated categories based upon a metric further comprises:

25 said metric being based on the testing of a Boolean variable, said Boolean variable associated with a memory attribute.

44. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a set of predefined categories is
30 activated, a mechanism for selecting categories and executing actions associated with said categories, said mechanism comprising:

means for defining a list of categories activatable by said program;

means for identifying a first set of categories activated by said input;

means for selecting a second set of categories from said first set of activated categories based upon a metric, said metric computed based on the conditions located within each said activated category;

means for executing actions associated with said second set of categories.

45. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a set of predefined categories is activated, a method for selecting categories and executing actions associated with said categories, the steps of said method comprising:

(a) defining a list of categories activatable by said program;

(b) for an input,

(i) selecting a set of activated categories based upon a metric, said metric computed based on the conditions located within each said activated category;

(ii) executing actions associated with said set of categories.

46. The method for selecting categories and executing actions associated with said categories as recited in claim 45, wherein the step of selecting a set of activated categories based upon a metric further comprises:

said metric computed based on conditions in categories such that said category is activated if said condition is satisfied by said input.

47. The methods of selecting categories and executing actions associated with said categories as recited in claim 46, wherein the step of selecting a set of activated categories based upon a metric further comprises:

(i) constructing a graph in which the nodes represent predetermined states of user input and the edges represent user input transitions between said nodes;

(ii) associating a set of final nodes with said conditions within categories, wherein one of said conditions is satisfied if transitions upon a given user input place the state of user input at the associated final node.

48. The method for selecting categories and executing actions associated with said categories as recited in claim 47, wherein the step of selecting a set of activated categories based upon a metric further comprises:

5 said metric based on the length of the path in the graph traversed to a final node of a condition.

49. The method for selecting categories and executing actions associated with said categories as recited in claim 48, wherein the step of selecting a set of activated categories based upon a metric further comprises:

10 selecting categories among said first set of categories for which the path length is maximized.

50. The method for selecting categories and executing actions associated with said categories as recited in claim 45, wherein the step of selecting a set of categories based upon a metric further comprises:

15 said metric based on the frequency with which said conditions are expected to be true.

51. The method for selecting categories and executing actions associated with said categories as recited in claim 50, wherein said metric further comprises:

- 20 (i) defining conditions to be patterns of user input;
 (ii) estimating the frequency of conditions occurring in user input.

52. The method for selecting categories and executing actions associated with said categories as recited in claim 51, wherein the step of estimating the frequency of conditions occurring in user input further comprises:

25 estimating the frequency of conditions occurring in example user inputs included in category scripts.

53. The method for selecting categories and executing actions associated with said categories as recited in claim 50, wherein the step of selecting a set of activated categories based upon a metric further comprises:

30 selecting categories for which the conditions associated with the categories are least likely to be true.

54. The method for selecting categories and executing actions associated with said categories as recited in claim 45, wherein the step of selecting a set of activated categories based upon a metric further comprises:

- 5 (i) associating metrics with the specificity of inputs supplied by a user;
(ii) selecting said categories whose metrics are associated with the most specific input currently supplied by said user.

55. The method for selecting categories and executing actions associated with said categories as recited in claim 45, wherein the step of selecting a set of activated categories based upon a metric further comprises:

said metric being a numeric value computed for each activated category, the value being based upon the frequency of matched words, partial words, and symbols found in the current input with words, partial words, and symbols found in the conditional clauses located within the category.

56. The method for selecting categories and executing actions associated with said categories as recited in claim 45, wherein the step of selecting a set of activated categories based upon a metric further comprises:

selecting said category having more matched words, partial words, and symbols in the conditional clause located within the category.

57. The method for selecting categories and executing actions associated with said categories as recited in claim 45, wherein the step of selecting a set of activated categories based upon a metric further comprises:

said metric being based on the testing of a Boolean variable, said Boolean variable associated with a memory attribute.

58. The method for selecting categories and executing actions associated with said categories as recited in claim 45, wherein the step of selecting a set of activated categories based upon a metric further comprises:

selecting a category from among said set of activated categories wherein two or more categories have the same computed metric.

59. The method for selecting categories and executing actions associated with said categories as recited in claim 58, wherein the step of selecting a category from among a set of activated categories wherein two or more categories have the same computed metric further comprises:

5 selecting the category for which the position on the focus of attention stack is highest.

60. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a set of predefined categories is activated, a method for selecting categories, the steps of said method comprising:

10 (a) defining a list of categories activatable by said program, said categories comprising conditions, said conditions comprising a specificity value computed according to possible user inputs;

(b) selecting a category from among a set of activated categories, said selected
15 category comprising a condition whose specificity value is maximal according to the current user input.

61. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a set of predefined categories is activated, a method for associating pronouns employed by users in conversation with words included in the automated interface program, the steps of said method comprising:

20 defining a list of categories activatable by said program;
associating word-pronoun pairs with said categories;
maintaining a map of word-pronoun pairs according to categories that are
25 activated in the course of the conversation.

62. The method for associating pronouns employed by users in conversation with words included in the automated interface program as recited in claim 61, wherein the step of associating word-pronoun pairs with categories further comprises:

30 associating subjects with categories;
associating word-pronoun pairs with subjects.

63. The method for associating pronouns employed by users in conversation with words included in the automated interface program as recited in claim 61, wherein the step of maintaining a map of word-pronoun pairs further comprises:

5 placing a word-pronoun pair associated with a category in the map when said category is focused.

64. The method for associating pronouns employed by users in conversation with words included in the automated interface program as recited in claim 62, wherein the step of maintaining a map of word-pronoun pairs further comprises:

10 placing a word-pronoun pair associated with a subject in the map when said subject is focused.

65. The method for associating pronouns employed by users in conversation with words included in the automated interface program as recited in claim 61, the steps of said method further comprising:

recording the user input;

modifying said user input by replacing pronouns found in user input according to said maintained map;

using said modified user input as the input to the automated interface program.

20

66. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a predefined set of categories is activated, a method for automatically verifying the performance of said program, the steps of said method comprising:

- 25 (a) specifying inputs under which the program should be tested;
- (b) associating said inputs with conditions within categories in the program, each said condition comprising at least one response which could be given if said condition is satisfied;
- (c) executing said program under at least one said input;
- 30 (d) determining the correctness of the program response upon said input.

67. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein said inputs further comprise machine-readable text.

5 68. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of associating said inputs with conditions within categories further comprises:

embedding said inputs within the script defining said category.

10 69. The method for automatically verifying the performance of an automated interface program as recited in claim 68, wherein the step of embedding said inputs further comprises:

embedding said inputs within condition blocks within said category.

15 70. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of associating said inputs with conditions within categories further comprises:

embedding an input into an example statement within said category.

20 71. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of associating said inputs with conditions within categories further comprises:

associating multiple inputs with a condition by embedding said inputs into a plurality of example statements.

25 72. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of associating said inputs with conditions within categories further comprises:

- 30 i) for a given input, identifying analogous inputs that should produce the same response as said given input;
- ii) embedding said analogous inputs within example statements.

73. The method for automatically verifying the performance of an automated interface program as recited in claim 72, wherein the step of embedding said analogous inputs further comprises:

placing said example statements outside of said script defining said category.

5

74. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of associating said inputs with conditions within categories further comprises:

i) for a given input, identifying analogous inputs that should produce the same response as said given input, and wherein said analogous inputs are executed only when said associated category is focused;

ii) embedding said analogous inputs within example statements, said example statements being executed only when said associated category is focused.

75. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of associating said inputs with conditions within categories further comprises:

i) identifying a set of initial inputs, said initial inputs to be tested prior to testing of any other inputs;

ii) embedding said initial inputs within scripts defining said categories.

76. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of associating said inputs with conditions within categories further comprises:

i) identifying a set of sequential inputs, said sequential inputs to be tested in order of input;

ii) embedding said sequential inputs within scripts defining said categories.

77. The method for automatically verifying the performance of an automated interface program as recited in claim 75, wherein the step of executing said program under at least one said input further comprises:

i) executing said initial inputs;

ii) saving the state of said program for the further execution of other inputs.

78. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of executing said program under at least one said input further comprises:

for each said input,

- i) saving said state of the program;
- ii) executing said input;
- iii) restoring said state of the program.

79. The method for automatically verifying the performance of an automated interface program as recited in claim 77, wherein the step of executing said program under at least one said input further comprises:

for each said other input,

- i) saving said state of the program;
- ii) executing said other input;
- iii) restoring said state of the program.

80. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of determining the correctness of the program response upon said input further comprises:

determining whether the associated condition is satisfied upon said input.

81. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of determining the correctness of the program response upon said input further comprises:

determining whether the response associated with said condition is given upon said input.

82. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of determining the correctness of the program response upon said input further comprises:

- i) determining whether the associated condition is satisfied upon said input;
- and
- ii) determining whether the response associated with said condition is given upon said input.

83. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of determining the correctness of the program response upon said input further comprises:

5 identifying an error condition when said associated condition is not satisfied upon said input.

84. The method for automatically verifying the performance of an automated interface program as recited in claim 66, wherein the step of determining the correctness of the program response upon said input further comprises:

10 identifying an error condition when a response other than that associated with said condition is given upon said input.

85. A method for testing a program designed to converse with users in natural language. the steps of said method comprising:

- 15
- a) identifying natural language topics;
 - b) identifying text inputs that are associated with said natural language topics;
 - c) embedding executable statements based on said text inputs within scripts
- 20 embodying said natural language topics.

86. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a predefined set of categories is activated, a method for testing the context of an input when selecting a set of categories to activate, the steps of said method comprising:

- 25
- (a) associating subjects with categories;
 - (b) maintaining a list of the subjects associated with the most recently activated set of categories;
 - (c) testing conditions in categories such that the condition is satisfied if the
- 30 category comprises at least one subject in common with said list of subjects;
- (d) activating a set of categories, said set comprising those categories whose conditions are satisfied in step (c)

87. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 86 wherein step (a) further comprises:

associating subject keywords with categories by including said keywords in the script language defining said categories.

5

88. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 87 wherein step (a) further comprises:

creating a data structure by compiling said categories, said data structure comprising an association of subject keywords with said categories.

10

89. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 86 wherein step (b) further comprises:

- i) upon a user input, finding the set of activated categories
- ii) finding the set of subject keywords associated with said set of categories found in step (i).

15

90. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 89 wherein step (b) further comprises:

- iii) if said set of subject keywords in step (ii) is empty, maintaining the list of subjects associated with the previous input.

20

91. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 89 wherein step (b) further comprises:

- iii) if said set of subject keywords in step (ii) is empty, deleting the list of subjects associated with the previous input.

25

92. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 89 wherein the step of finding the set of activated categories further comprises:

- finding the set of categories that are explicitly focused.

30

93. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 89 wherein the step of finding the set of activated categories further comprises:

finding the set of categories that are automatically focused.

94. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 92 wherein the step of finding the set of categories that are explicitly focused further comprises:

finding the set of categories that are focused by explicit reference to said categories.

95. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 92 wherein the step of finding the set of categories that are explicitly focused further comprises:

finding the set of categories that are explicitly focused by reference to subject keywords associated with said categories.

96. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 95 wherein the step of finding the set of subject keywords associated with said set of categories comprises:

including in said maintained list of subjects all subject keywords associated with said categories explicitly focused by reference to subject keywords.

97. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 95 wherein the step of finding the set of subject keywords associated with said set of categories comprises:

including in said maintained list of subjects subject keywords found only in said references to subject keywords.

98. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 86 wherein step (c) further comprises:

upon user input, processing said input through a set of categories and, for each category in said set and having said condition, testing said condition.

99. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 98 wherein said set of categories further comprises priority categories.

100. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 98 wherein said set of categories further comprises default categories.

5

101. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 98 wherein said set of categories further comprises standard categories.

10

102. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 98 wherein the step of testing conditions in said categories further comprises:

testing each said category having said condition in order of execution.

15

103. The method for testing the context of an input when selecting a set of categories to activate as recited in Claim 98 wherein the step of testing conditions in said categories further comprises:

- i) finding a set of categories for testing by applying a best-fit matching mechanism;
- ii) testing each category in said set found in step (i).

20

104. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a predefined set of categories is activated, a mechanism for testing the context of an input when selecting a set of categories to activate, said mechanism comprising:

25

- (a) means for associating subjects with categories;
- (b) means for maintaining a list of the subjects associated with the most recently activated set of categories;
- (c) means for testing conditions in categories such that the condition is satisfied if the category comprises at least one subject in common with said list of subjects;

30

(d) means for activating a set of categories, said set comprising those categories whose conditions are satisfied in step (c).

105. A mechanism for placing conditions within scripts defining categories that allow the BOT to determine whether a particular category is appropriate to the current conversation with a user, said mechanism comprising:

- (a) means for including said conditions in a BOT script defining a category, said conditions relating subject keywords to said category;
- (b) means for activating categories upon a user input according to whether said conditions are satisfied if a subject keyword associated with a category matches a list of subject keywords determined by the state of the conversation.

106. In an automated interface program designed to interact and communicate with users, said program executing actions when a category among a set of predefined categories is activated, a method for generating scripts for categories based upon learning from user inputs, said method comprising:

- defining a set of example labels;
- extracting a set of candidate features from user inputs;
- determining a subset of said candidate features which correspond to said example labels;
- generating scripts for categories based on said subset of candidate features.

107. The method for generating scripts for categories as recited in claim 106, wherein the step of defining a set of example labels further comprises:

- defining example labels based on properties of a raw data stream.

108. The method for generating scripts for categories as recited in claim 107, wherein said raw data stream comprises text of previous user conversations.

109. The method for generating scripts for categories as recited in claim 106, wherein the step of extracting a set of candidate features further comprises:

- reducing text input into a vector of word tokens.

110. The method for generating scripts for categories as recited in claim 109, wherein the step of extracting a set of candidate features further comprises:

- converting said word tokens into symbolic codes.

111. The method for generating scripts for categories as recited in claim 110, wherein said symbolic codes encode the position of a word token within text input.

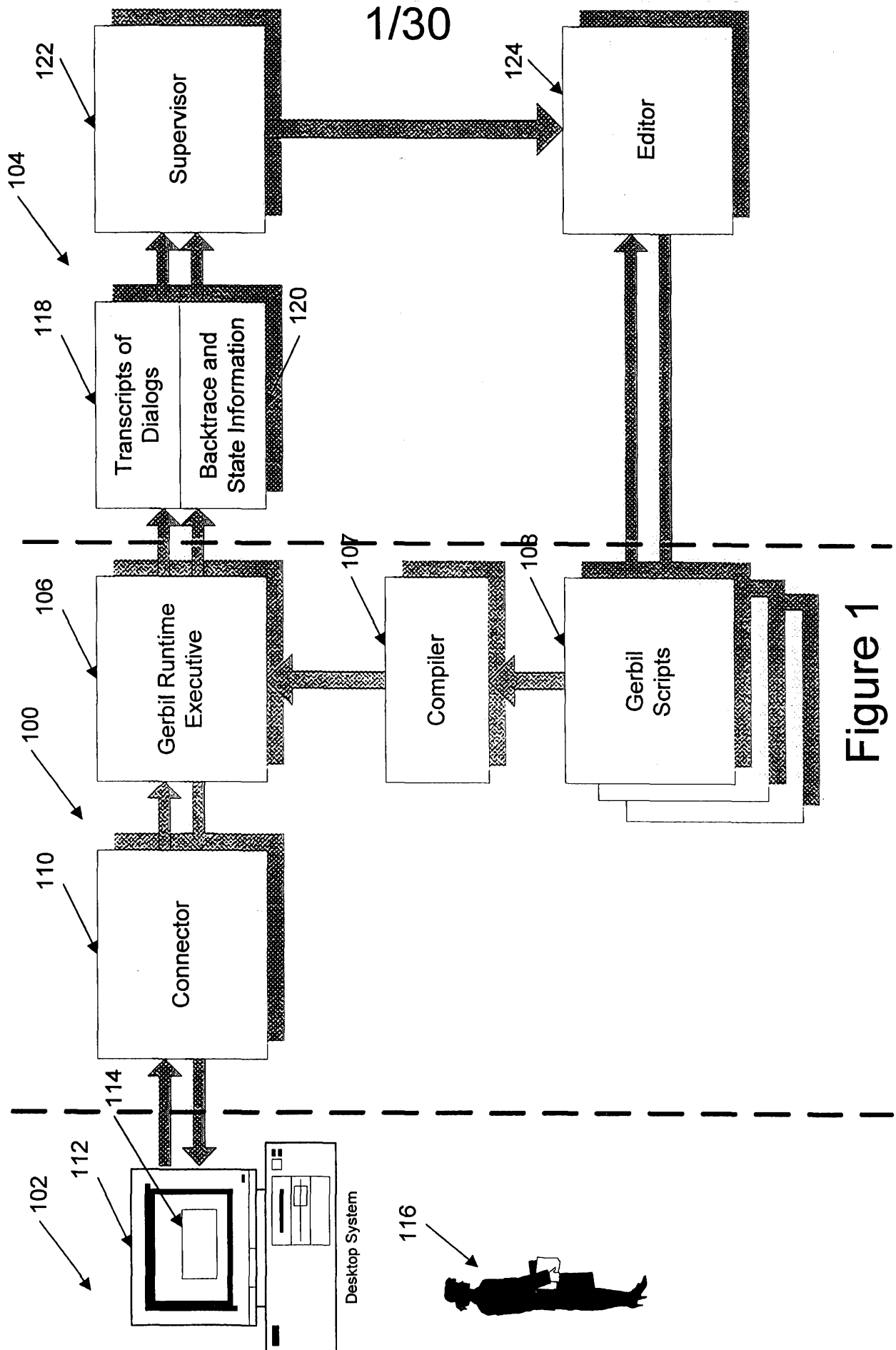
5 112. The method for generating scripts for categories as recited in claim 106, wherein the step of extracting a set of candidate features further comprises:
 entering features into a feature dictionary, such that said feature dictionary comprises a count of the number of times a feature occurs in user input.

10 113. The method for generating scripts for categories as recited in claim 106, wherein the step of determining a subset of said candidate features further comprises:
 determining estimated joint probability of occurrences of feature-label pairs.

15 114. The method for generating scripts for categories as recited in claim 113, wherein the step of determining a subset of said candidate features further comprises:
 computing conditional probabilities of labels on features and conditional probabilities of features on labels.

20 115. The method for generating scripts for categories as recited in claim 114, wherein the step of determining a subset of said candidate features further comprises:
 determining reliably predictive features from said conditional probabilities.

25 116. The method for generating scripts for categories as recited in claim 115, wherein the step of generating scripts for categories further comprises:
 generating scripts for categories based on said reliably predictive features.



2/30

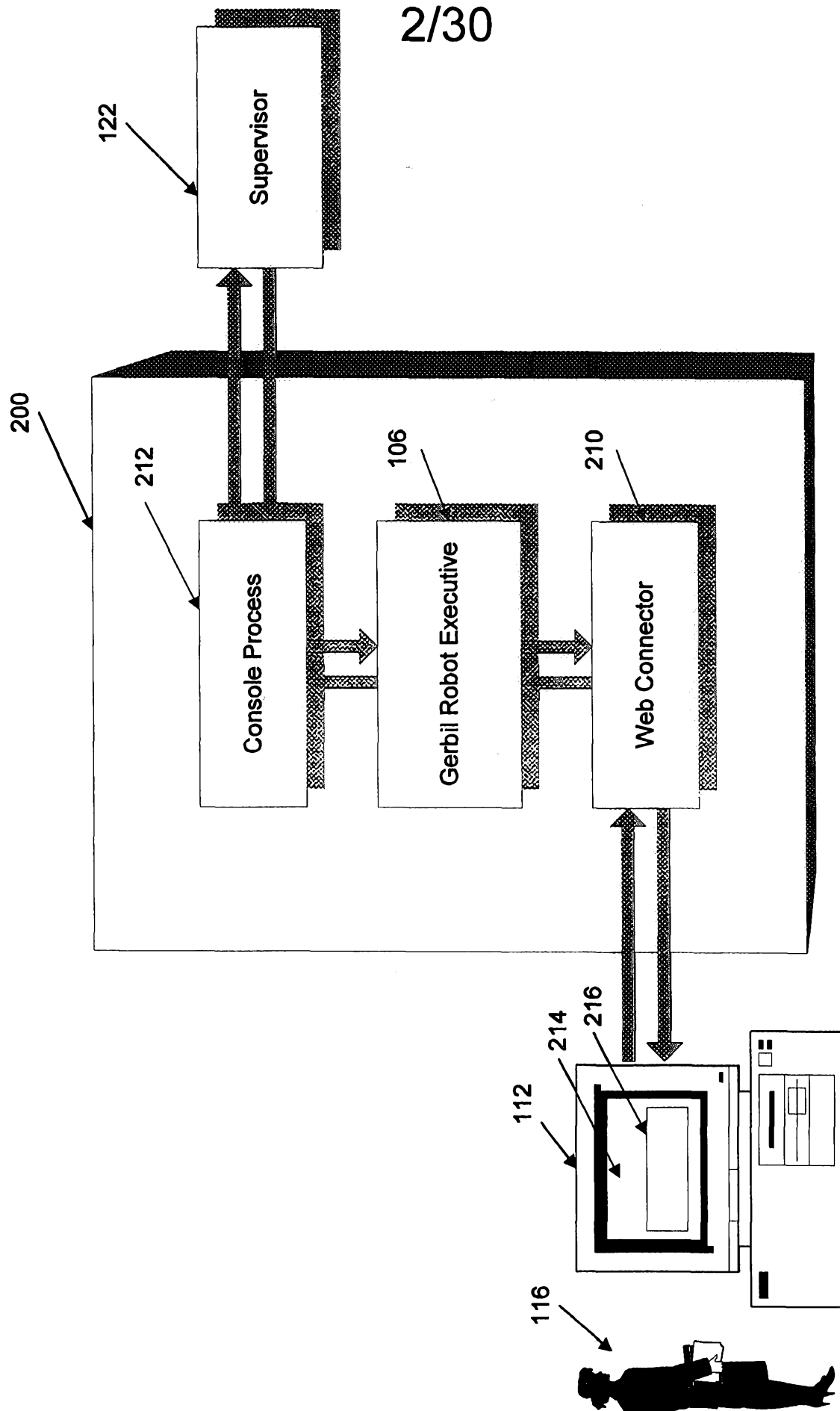


Figure 2

3/30

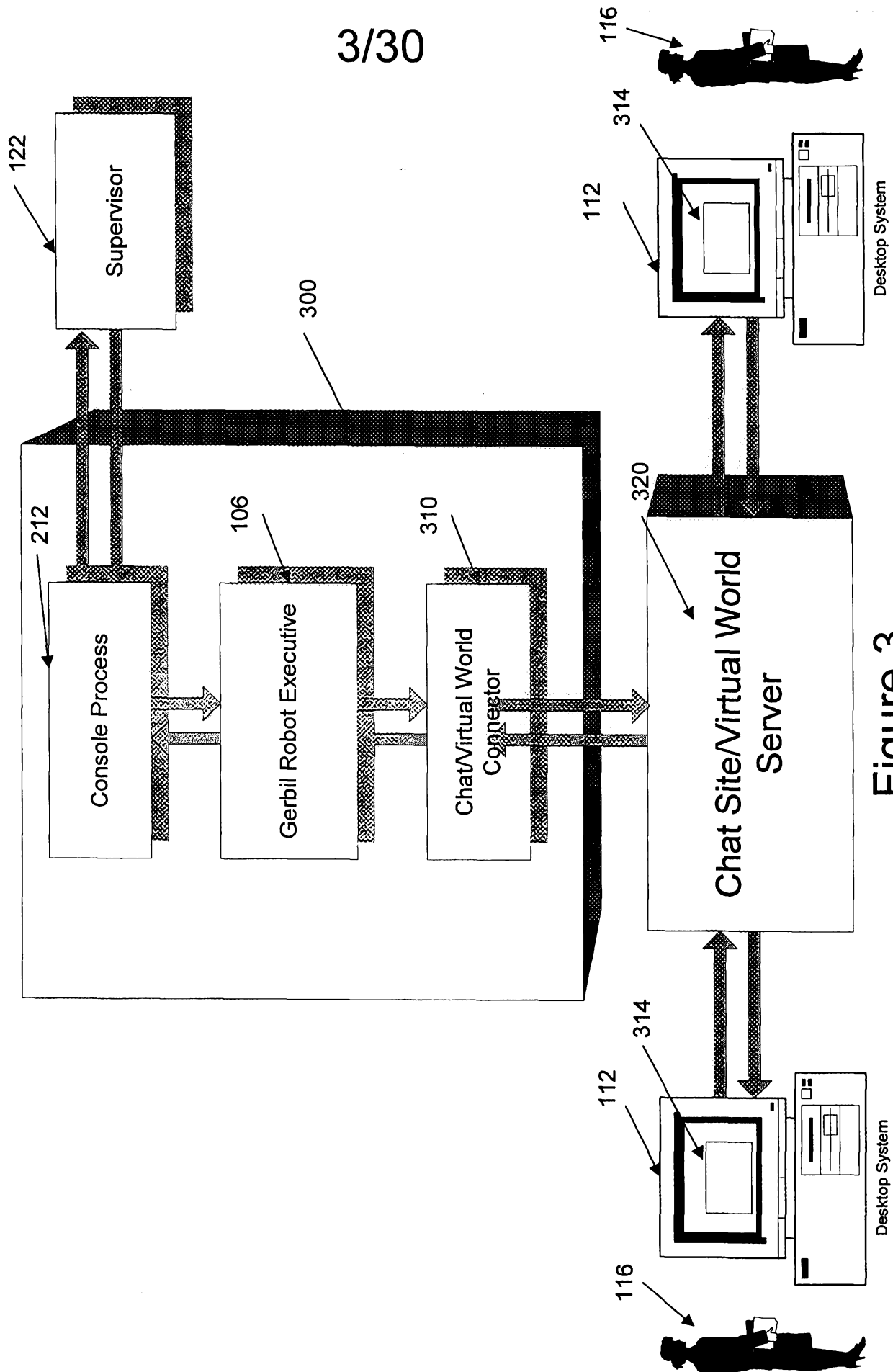


Figure 3

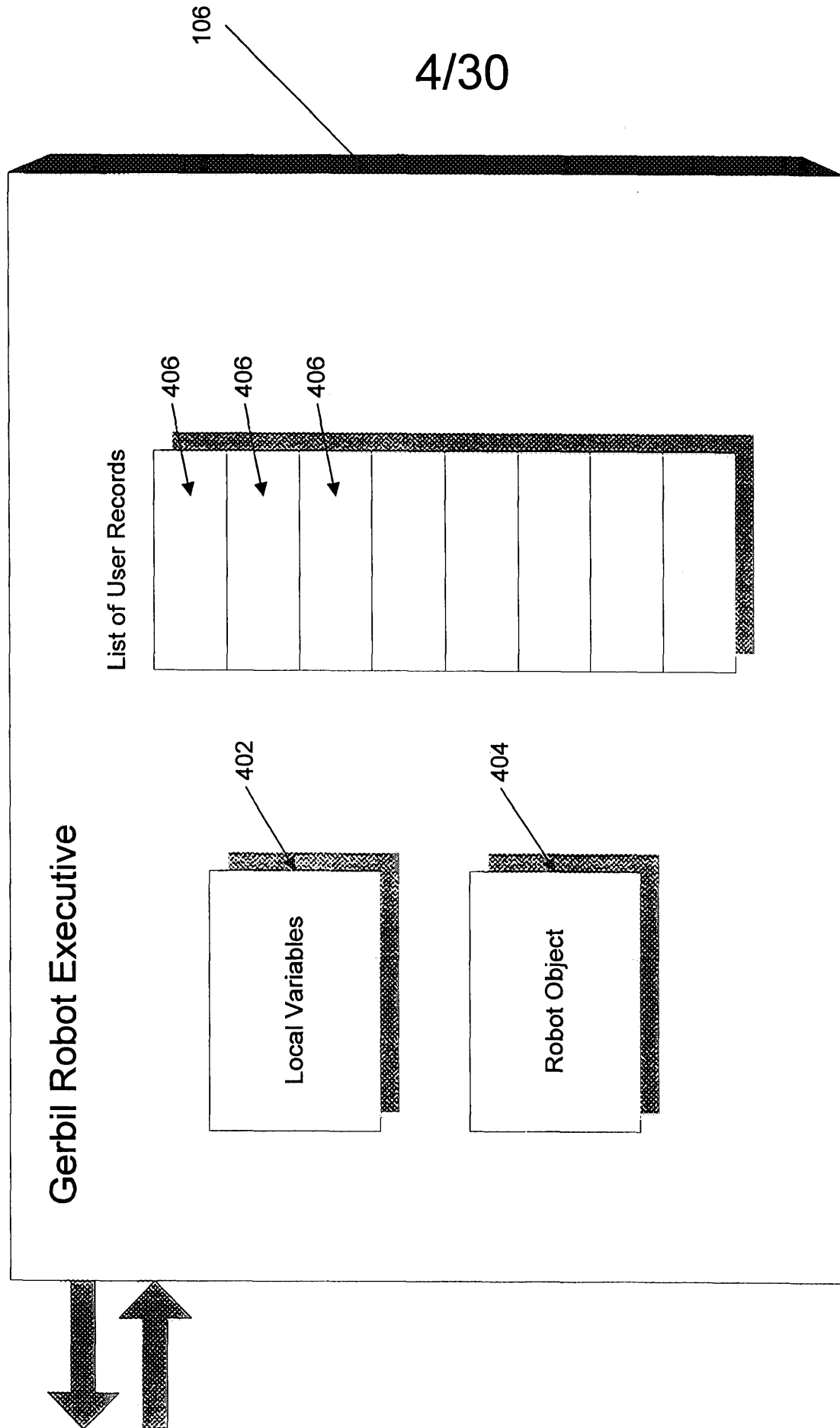


Figure 4

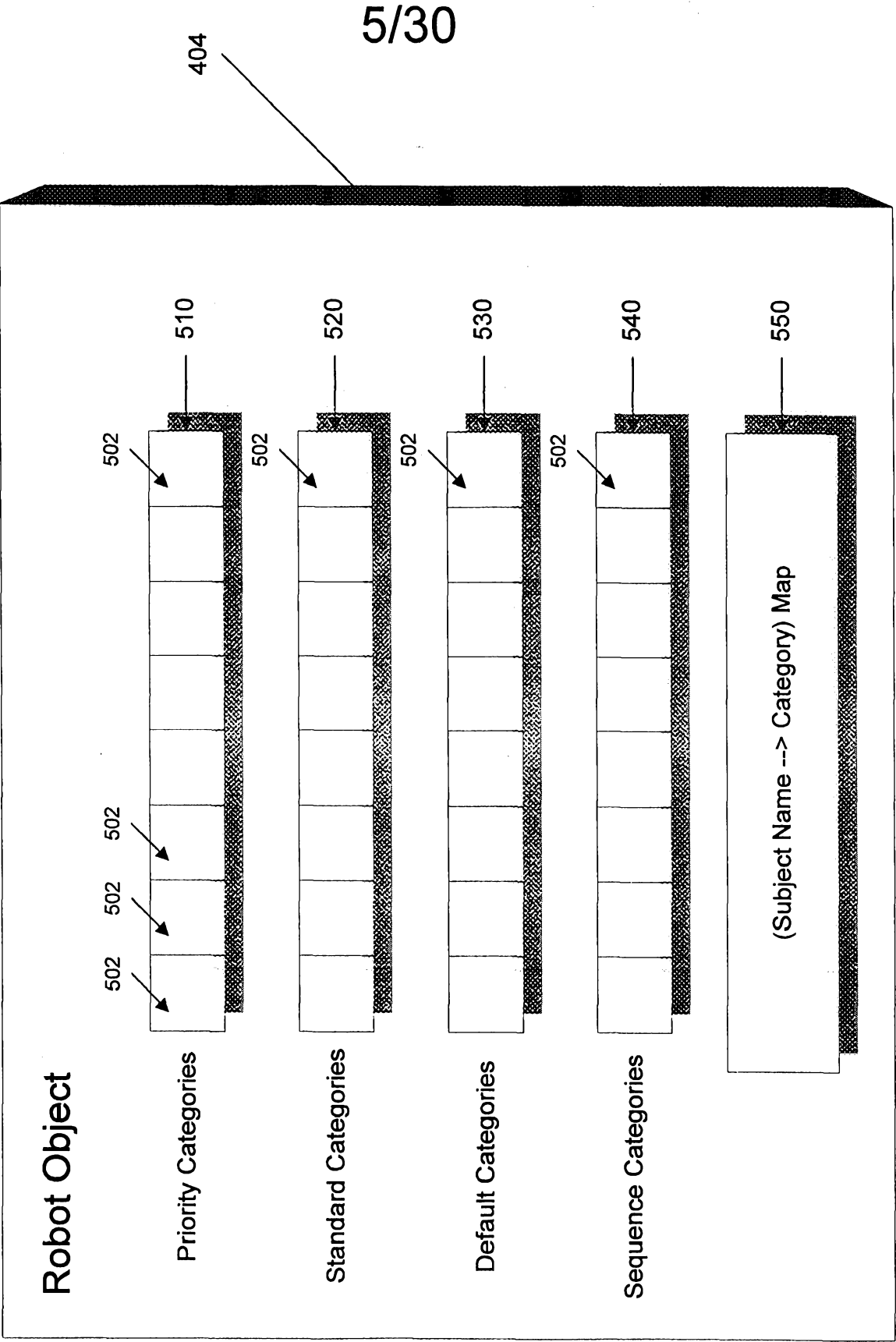


Figure 5

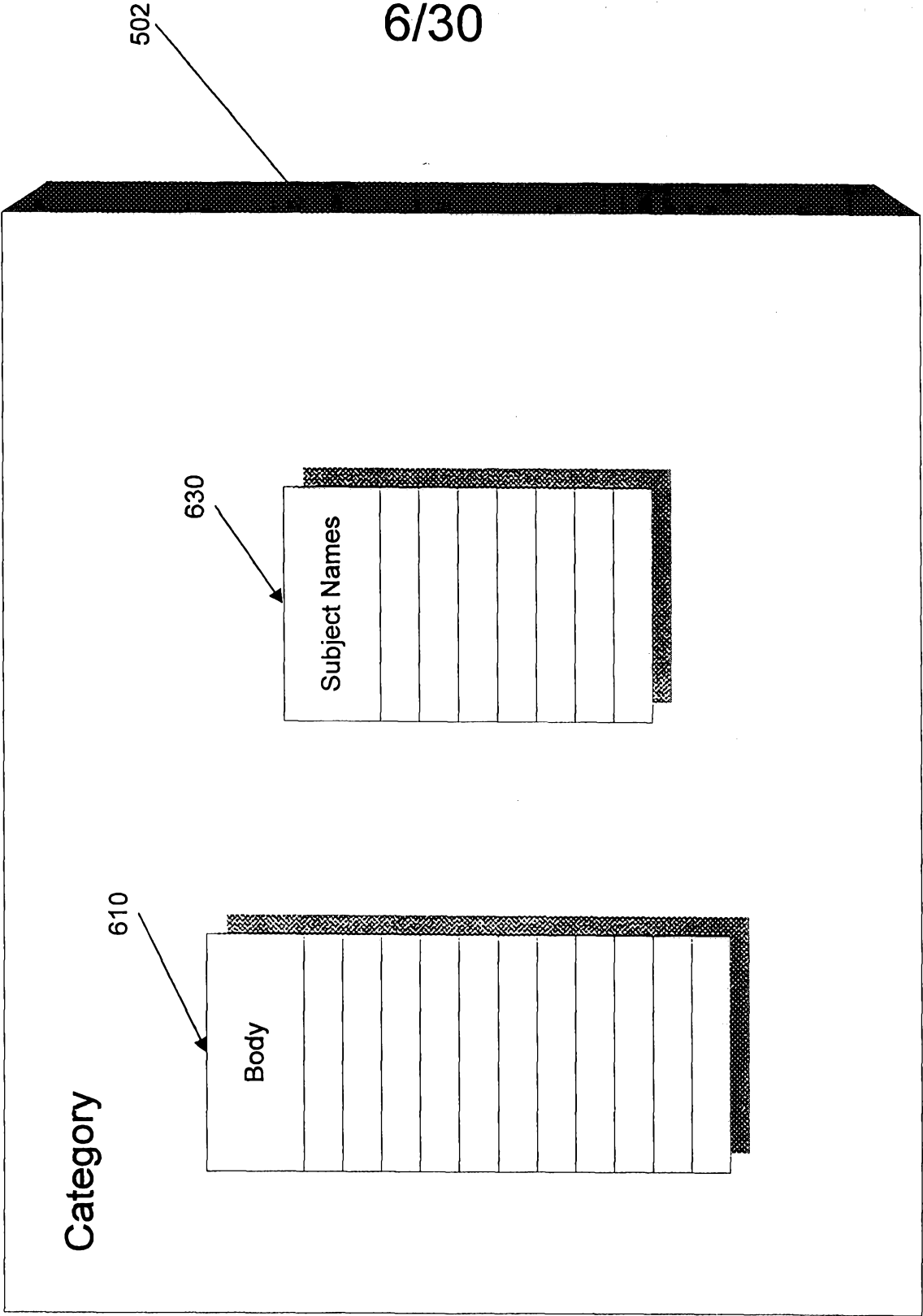


Figure 6

7/30

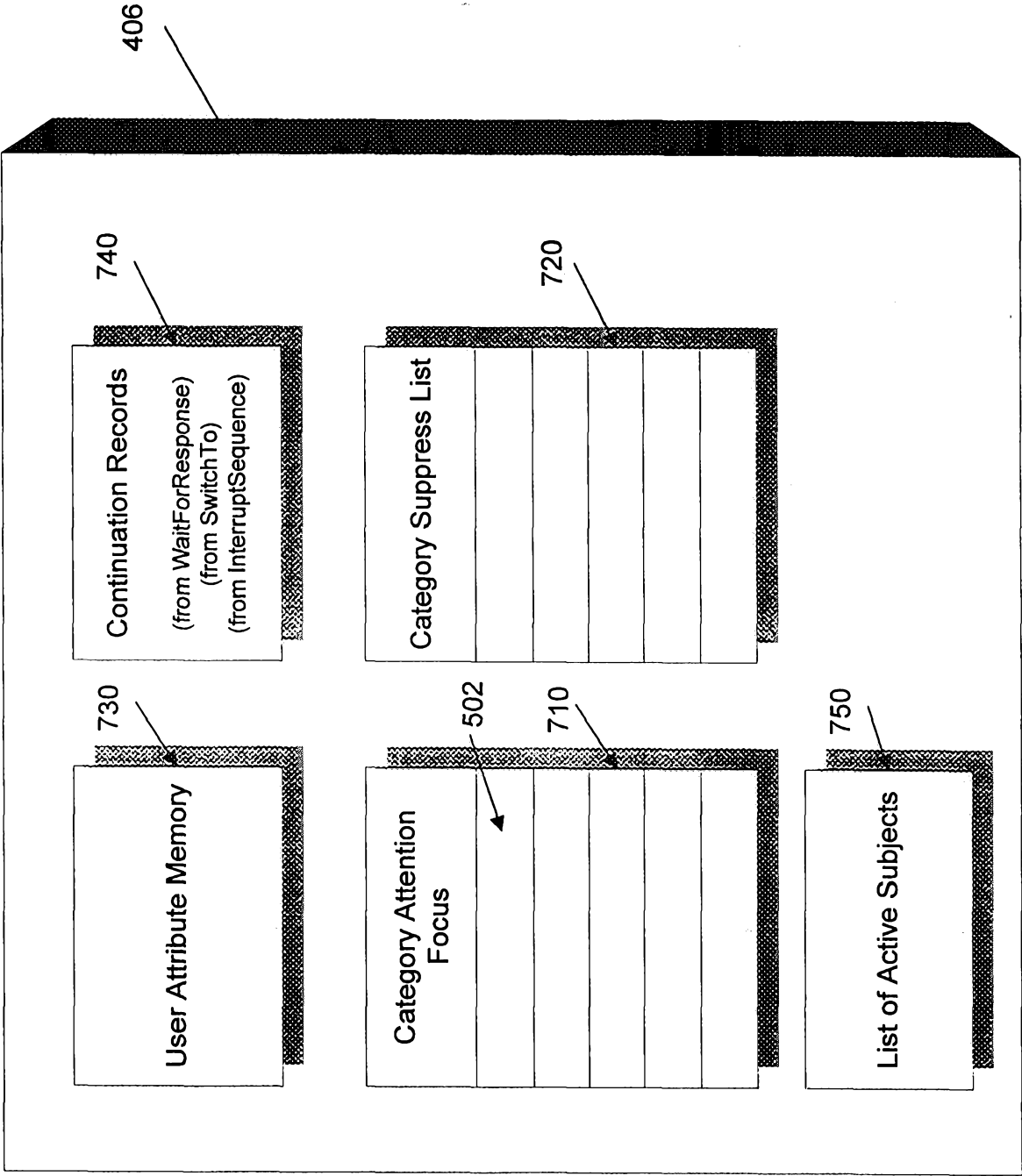


Figure 7

8/30

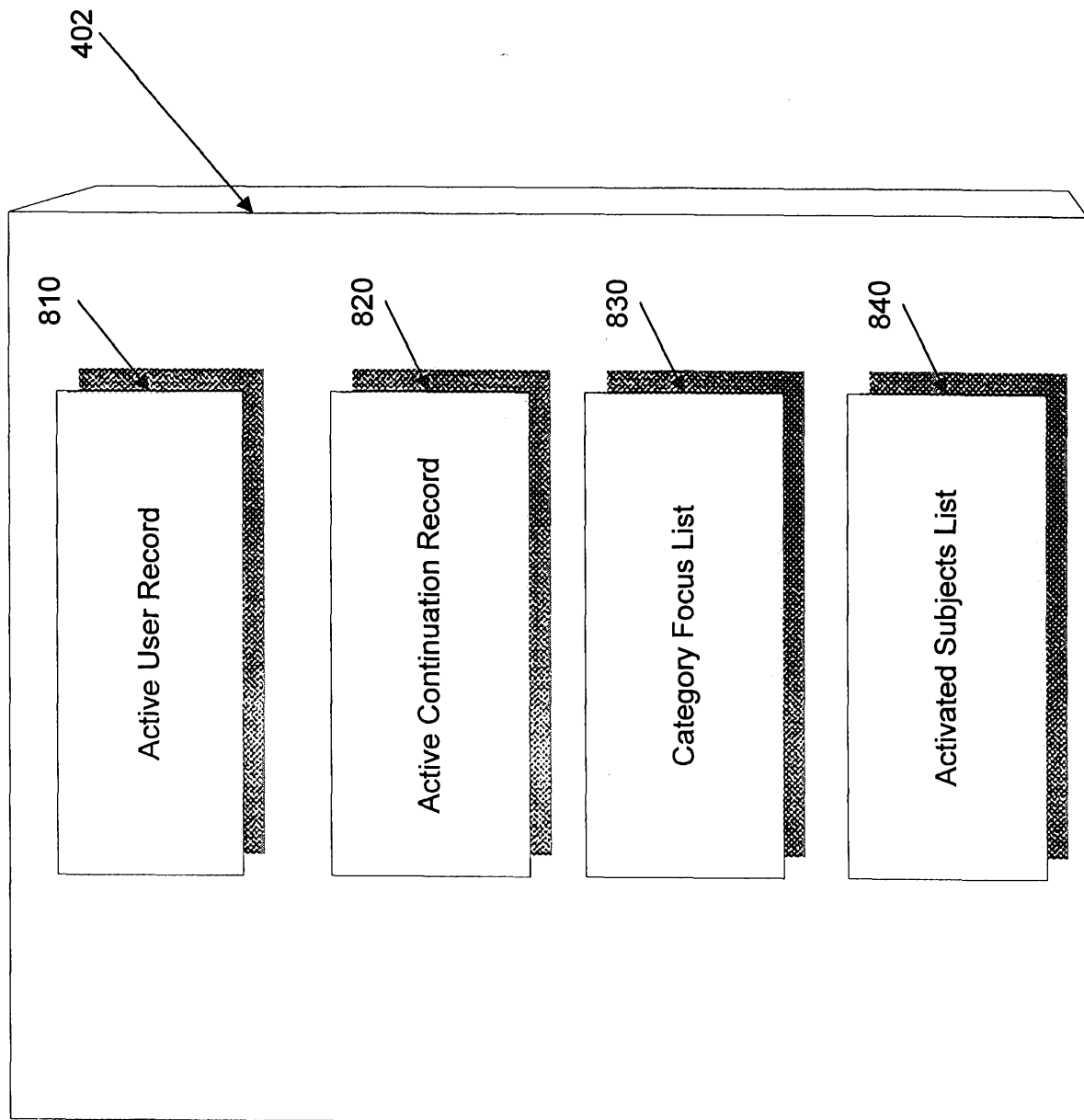


Figure 8

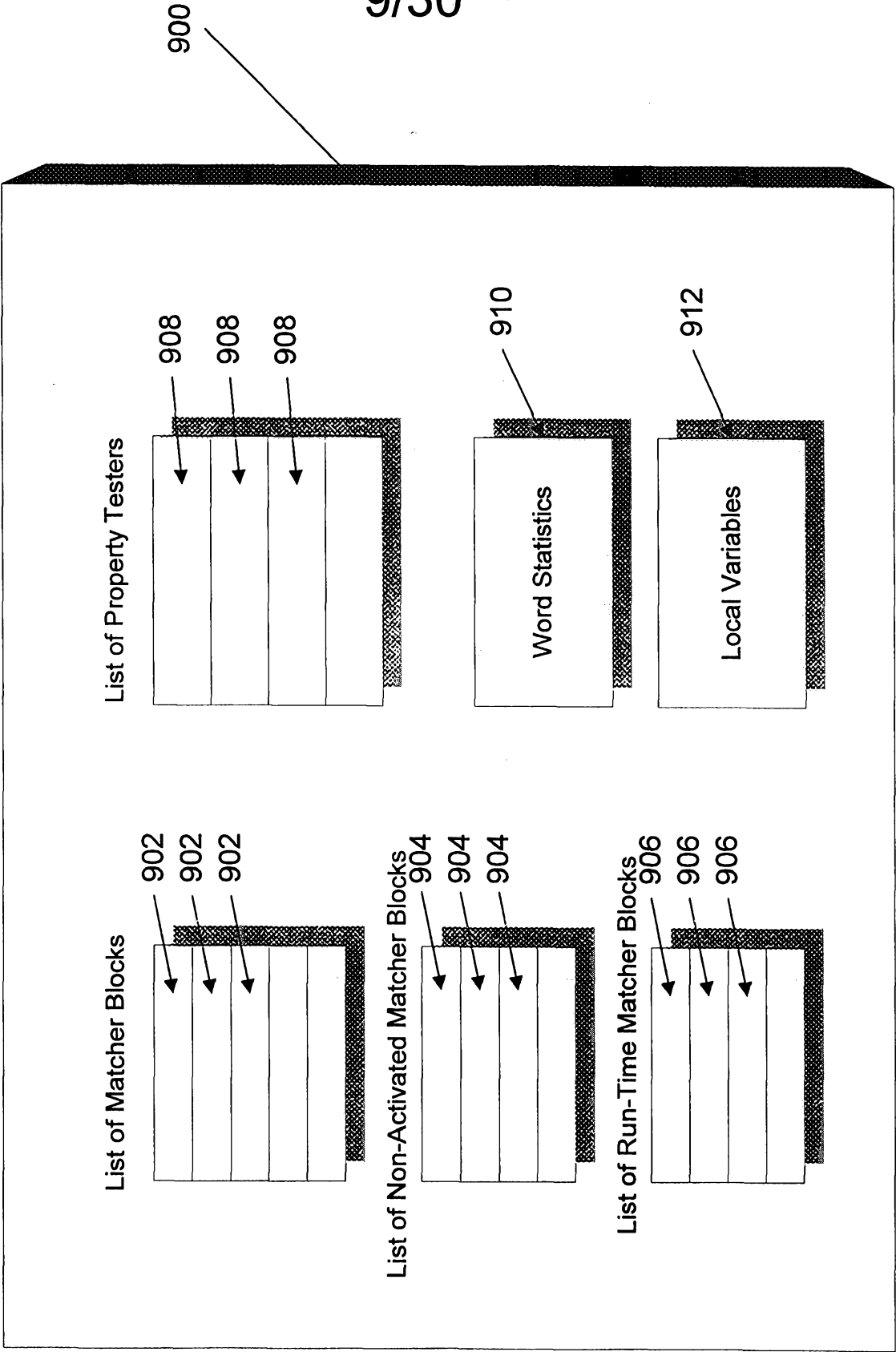


Figure 9

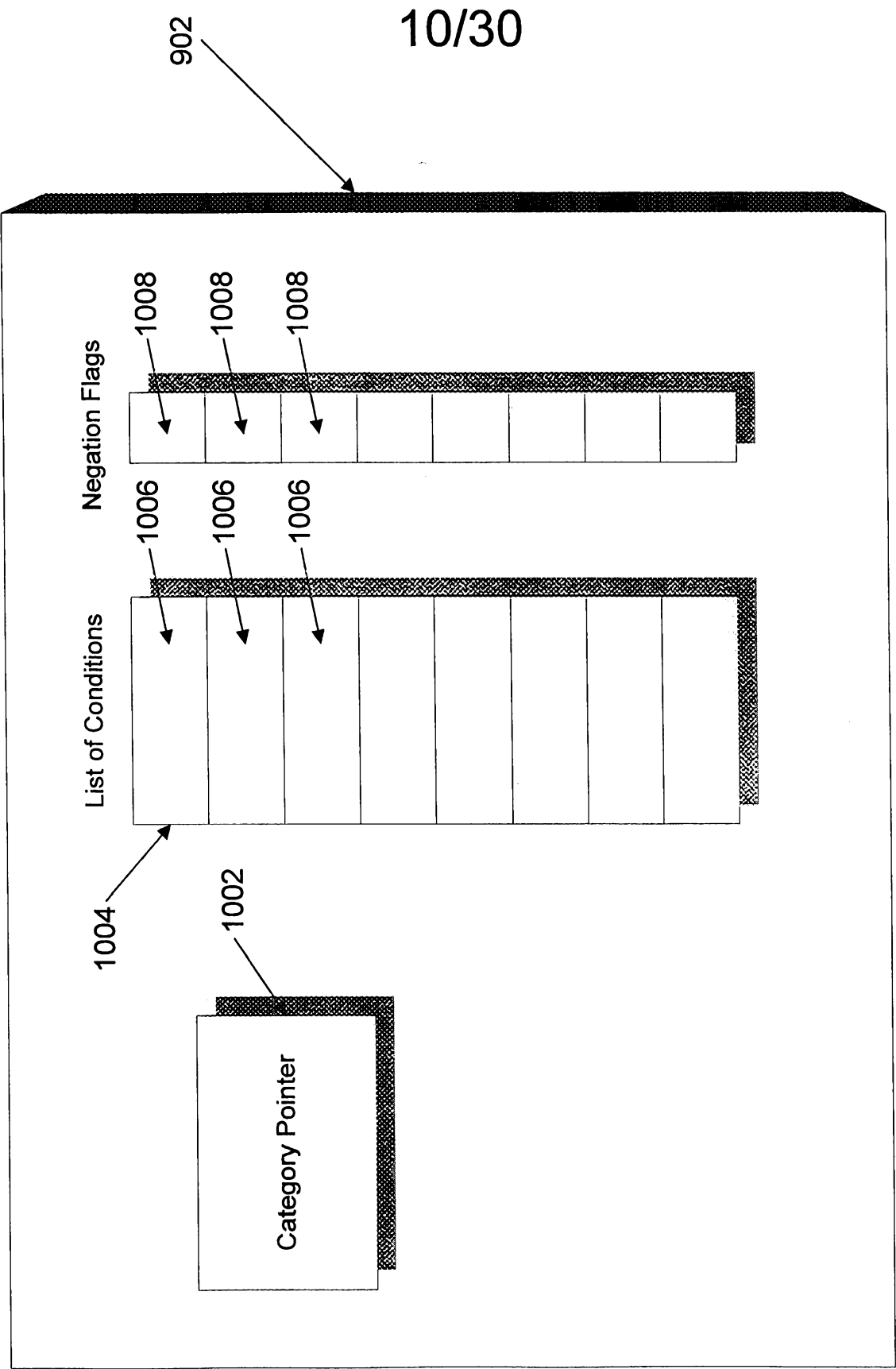


Figure 10

11/30

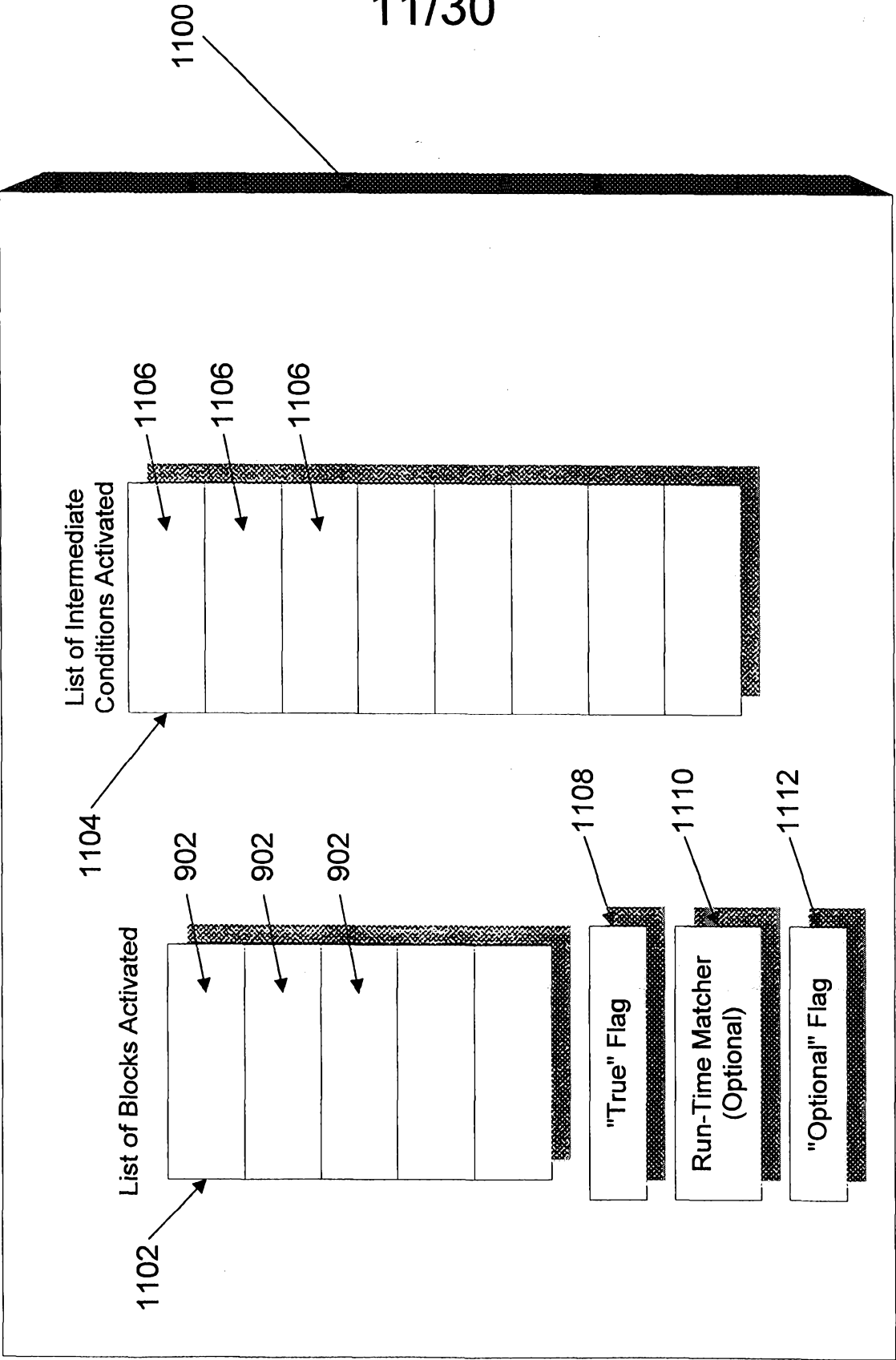


Figure 11

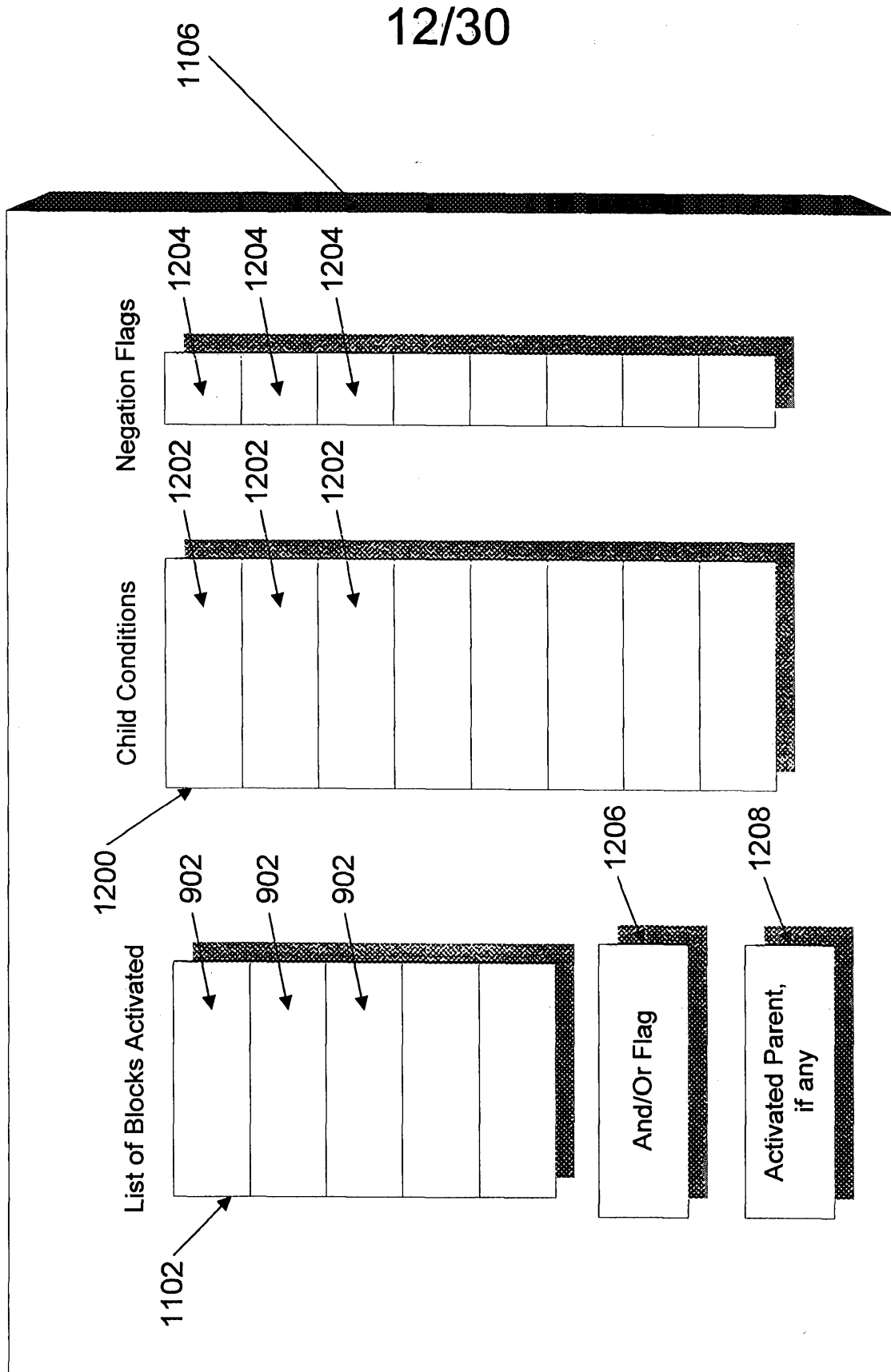


Figure 12

13/30

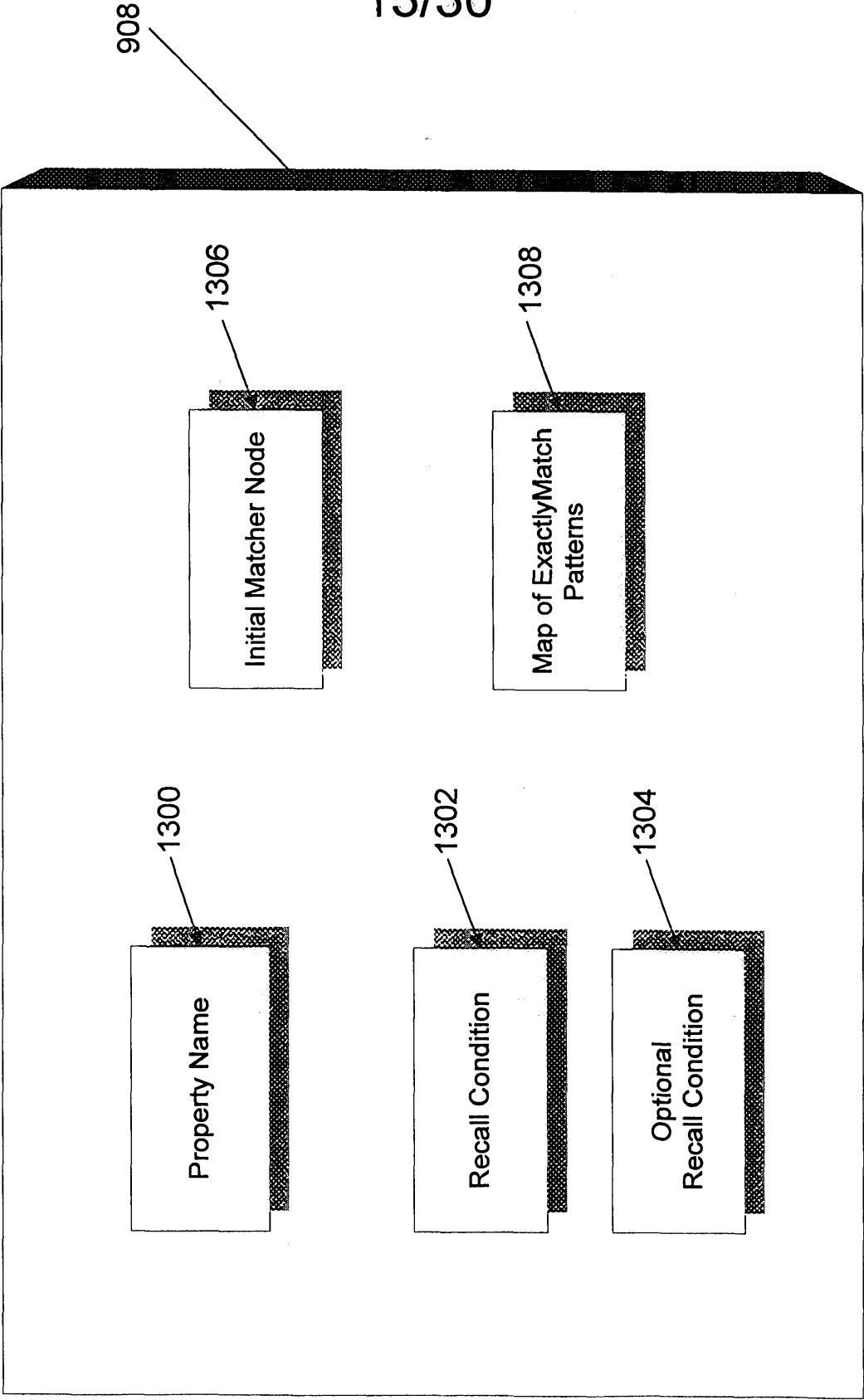


Figure 13

14/30

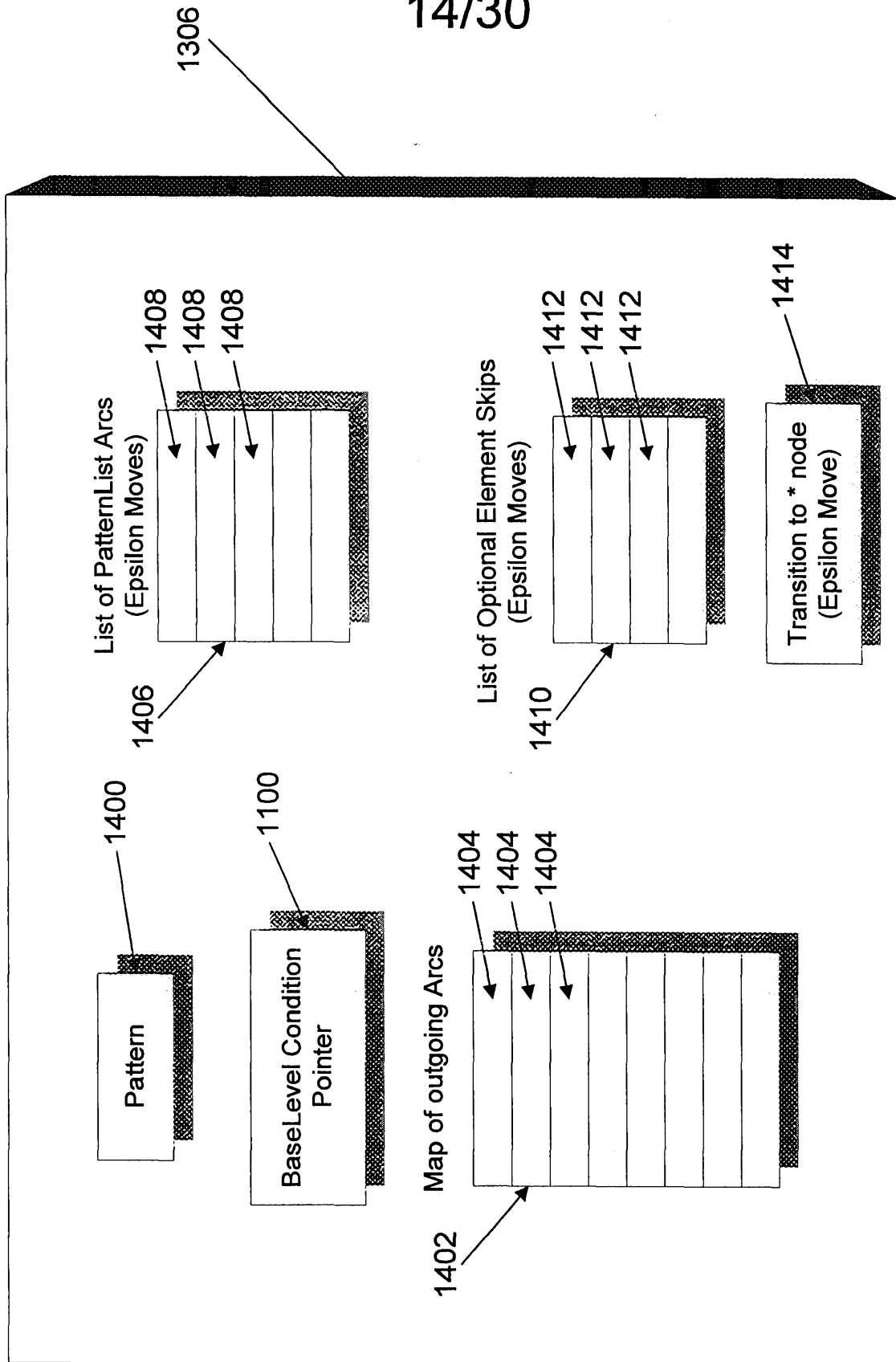


Figure 14

15/30

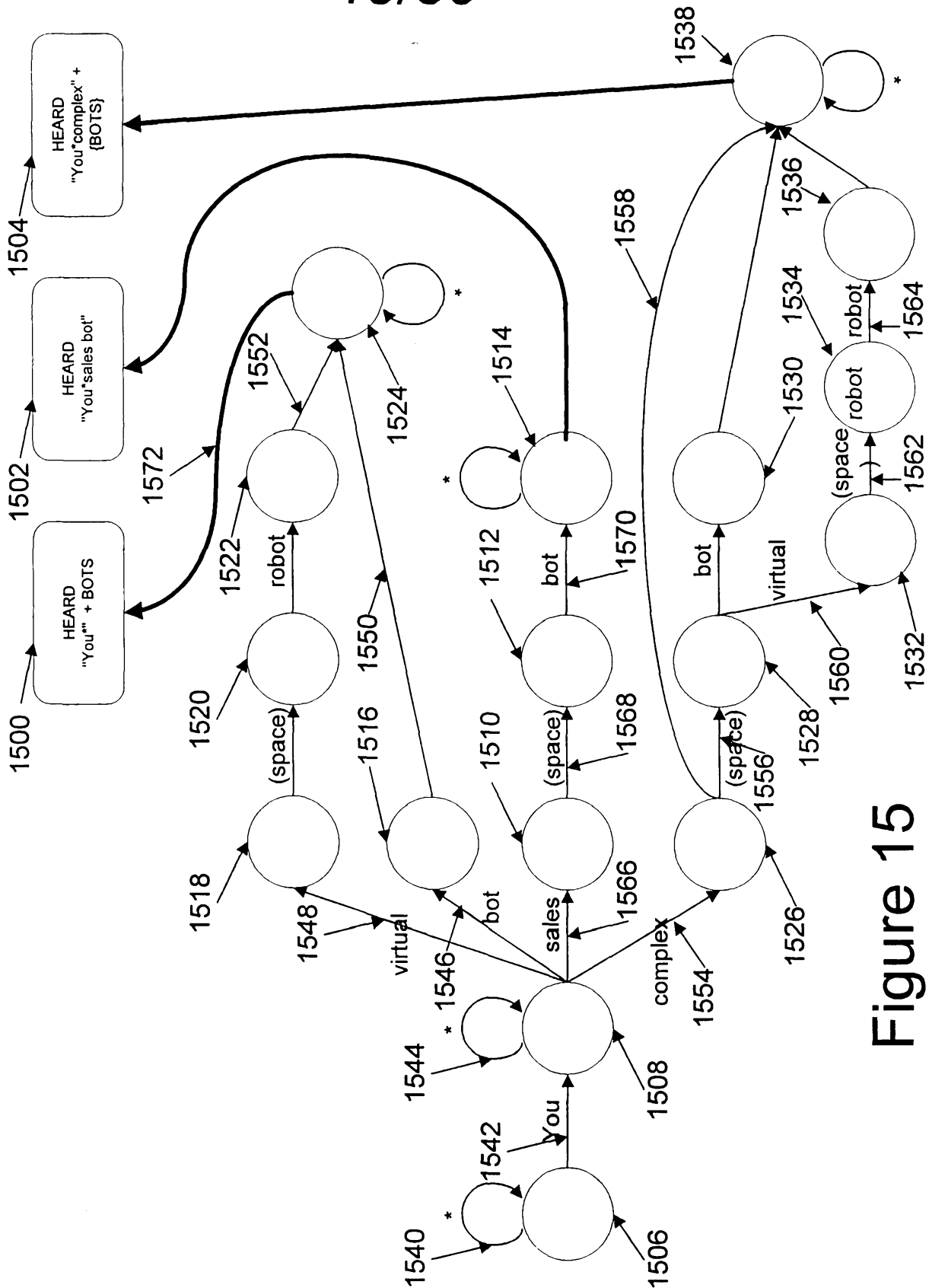


Figure 15

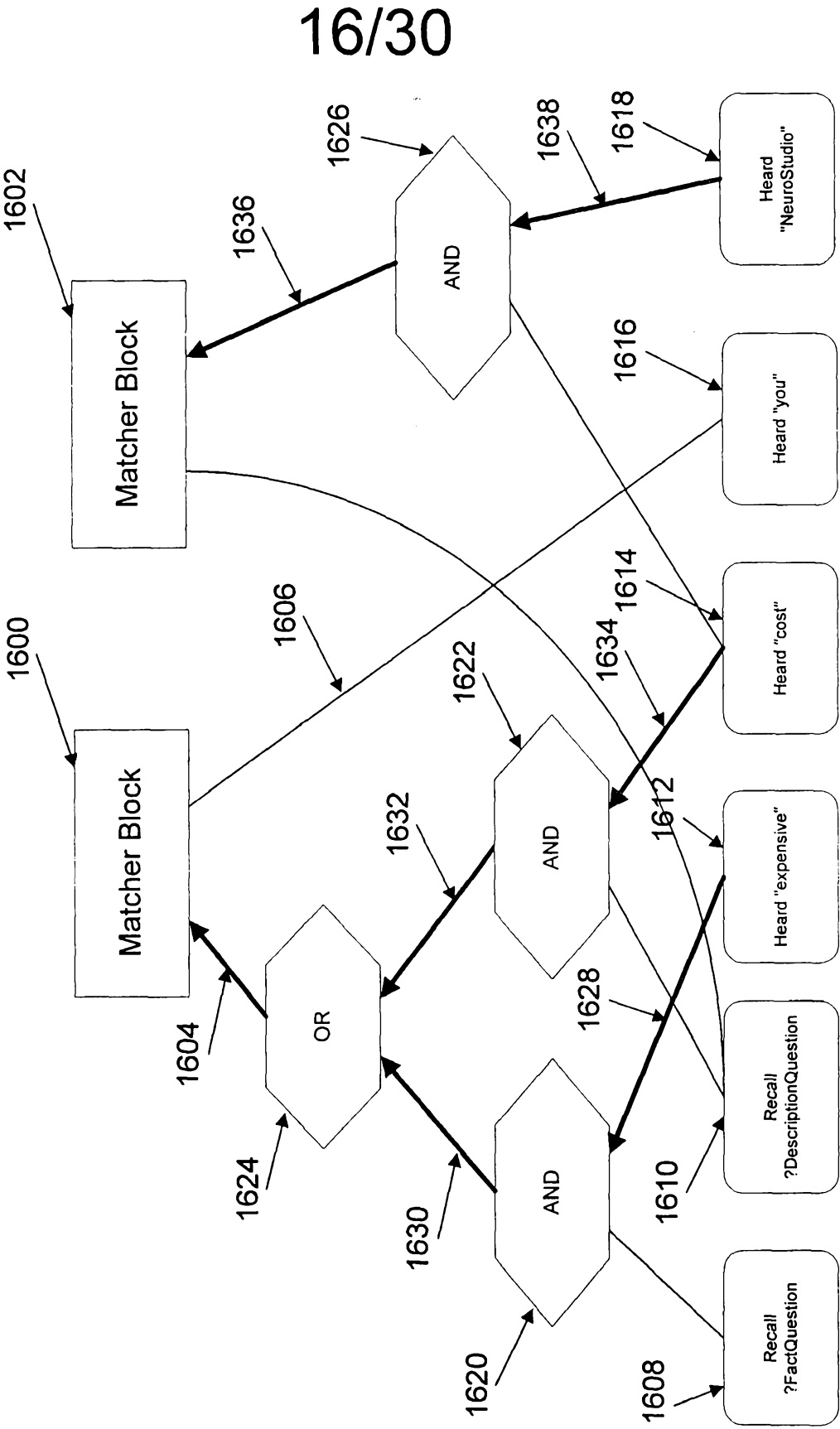


Figure 16

17/30

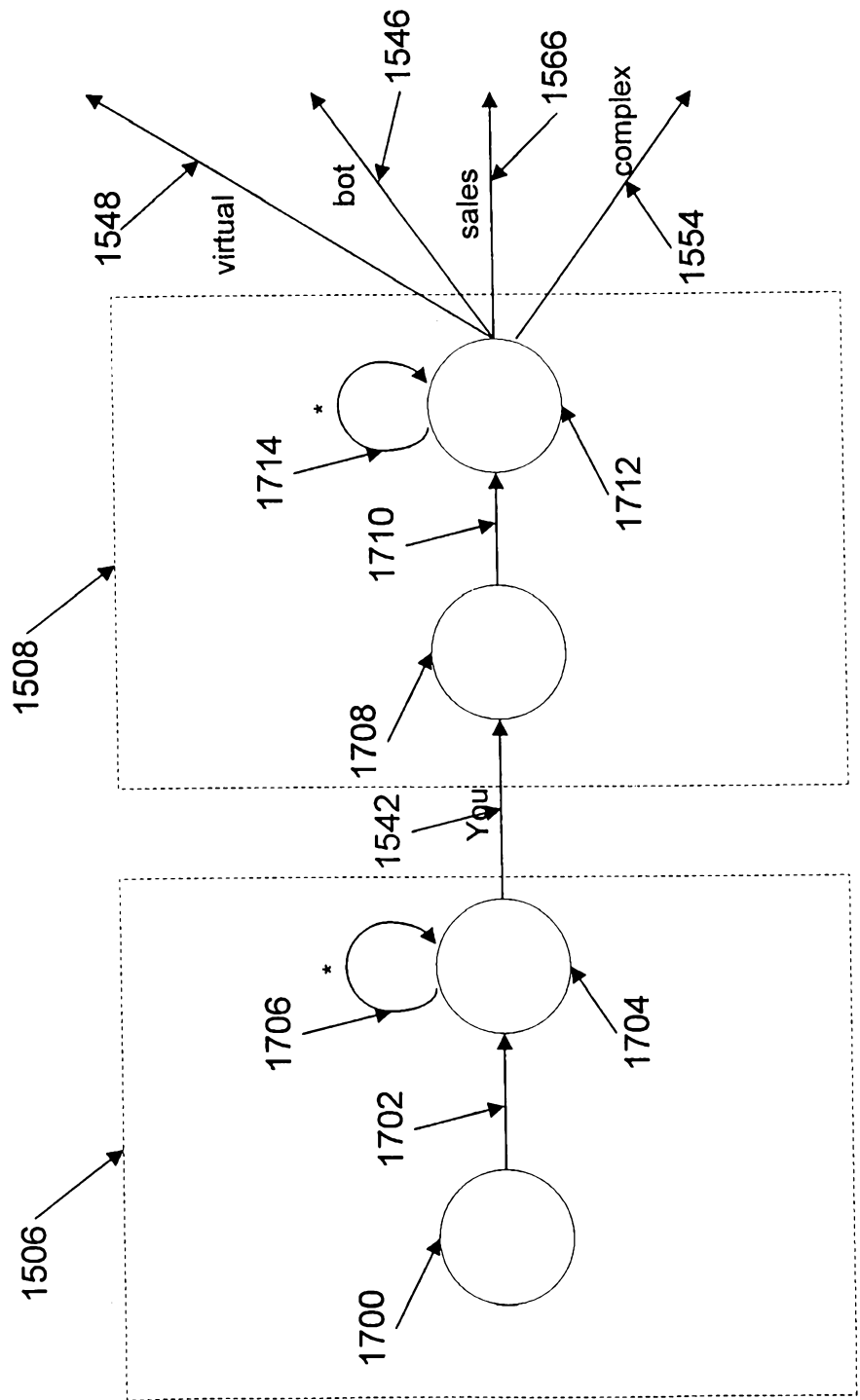


Figure 17

18/30

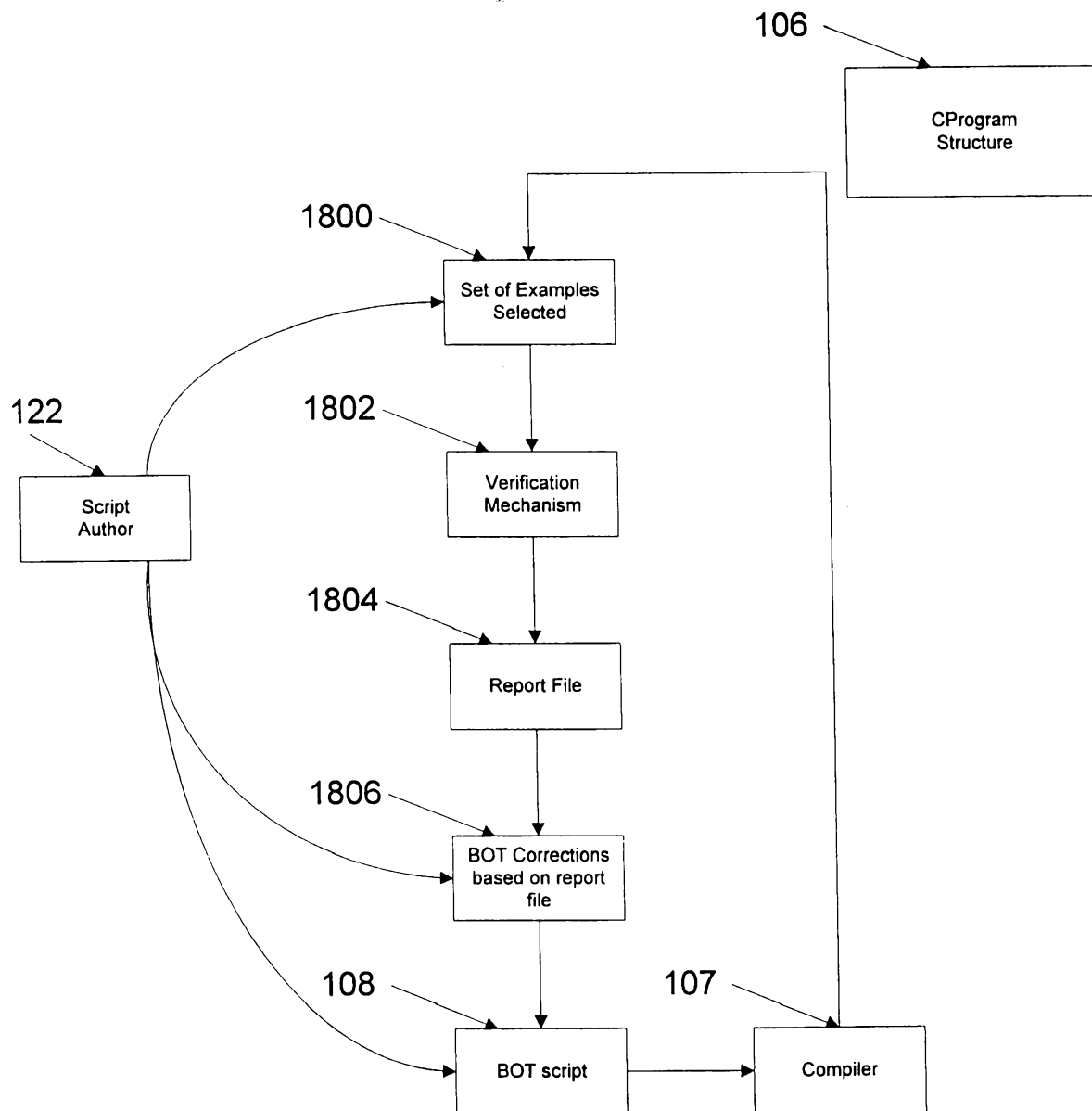


Figure 18

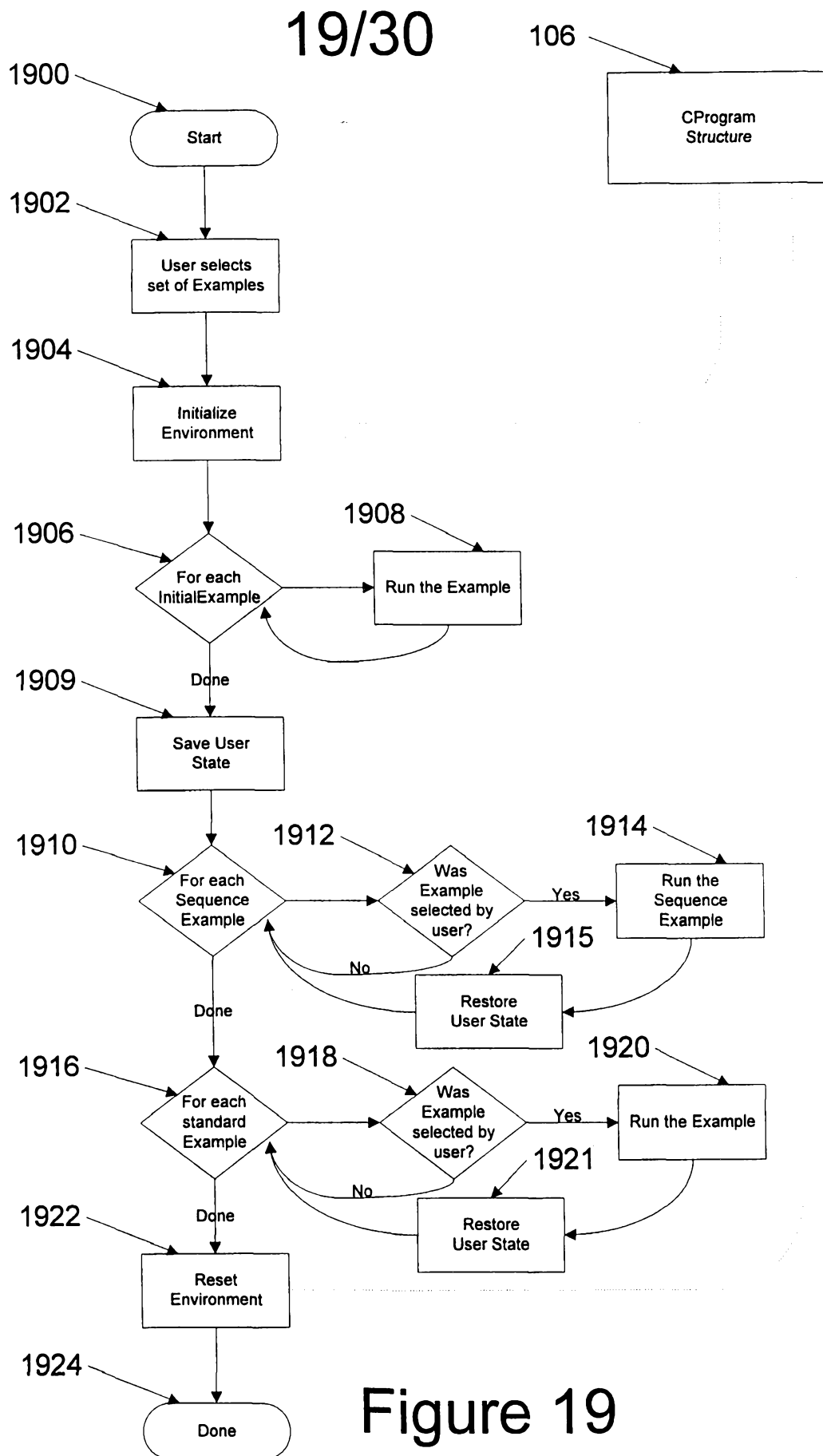


Figure 19

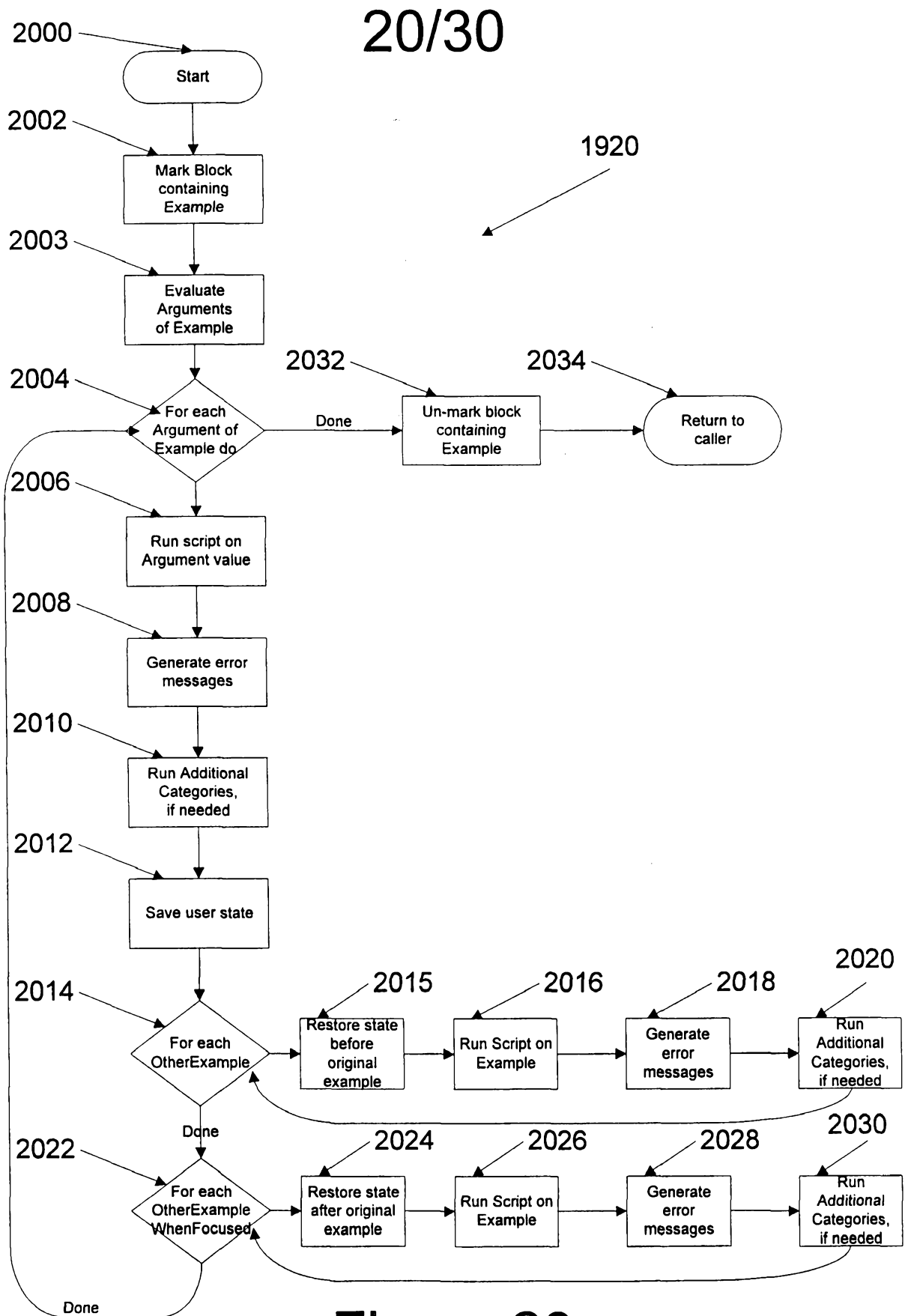


Figure 20

21/30

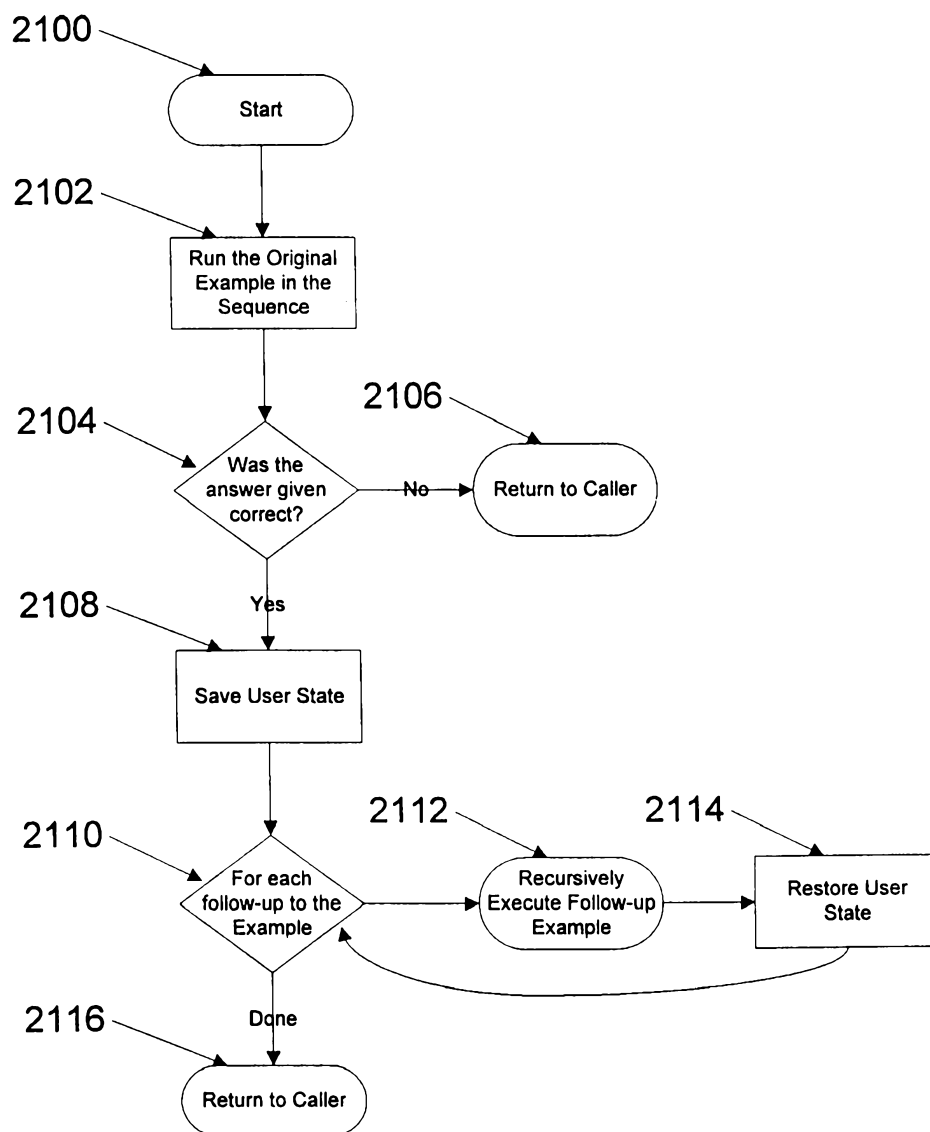


Figure 21

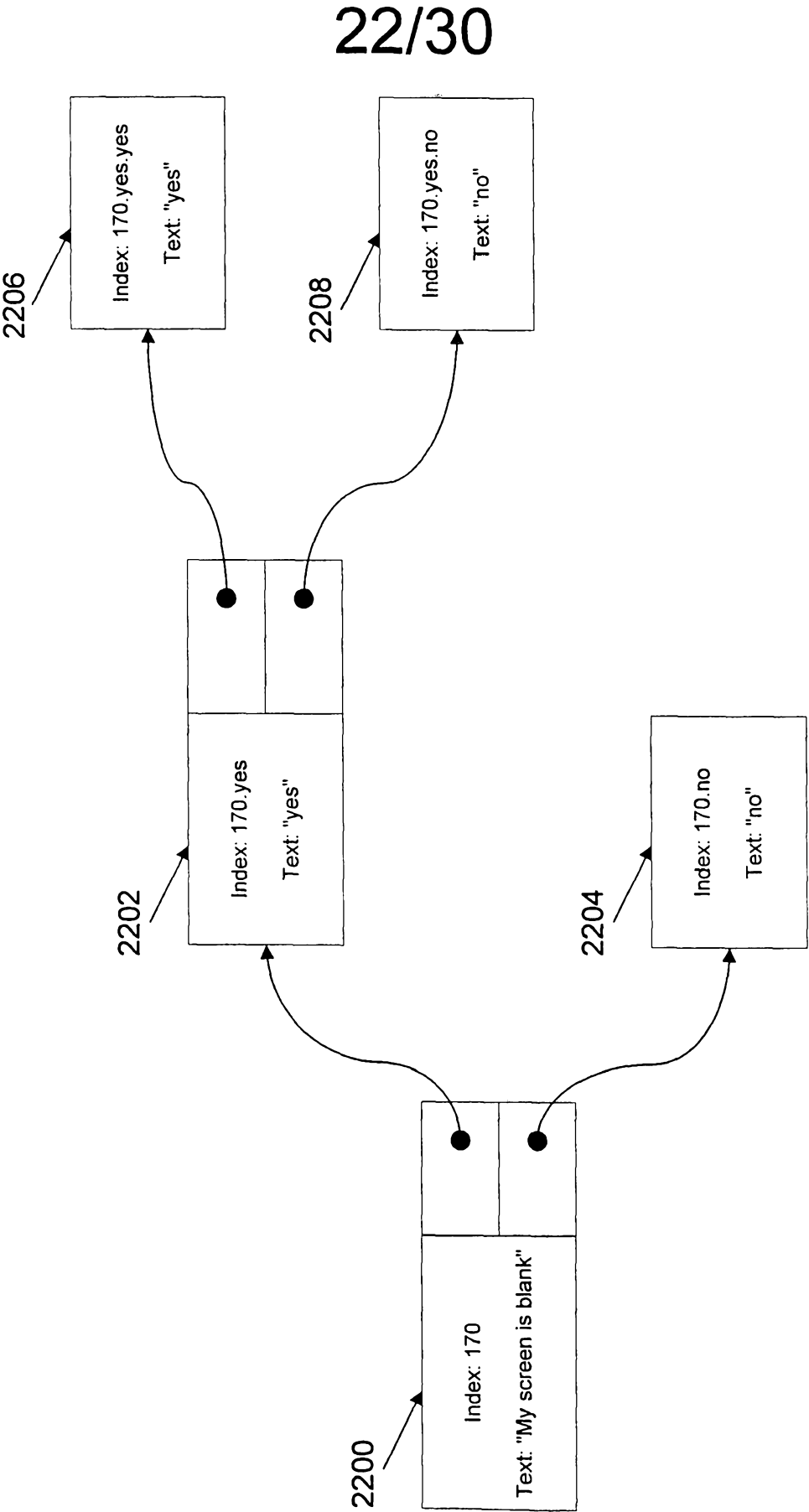


Figure 22

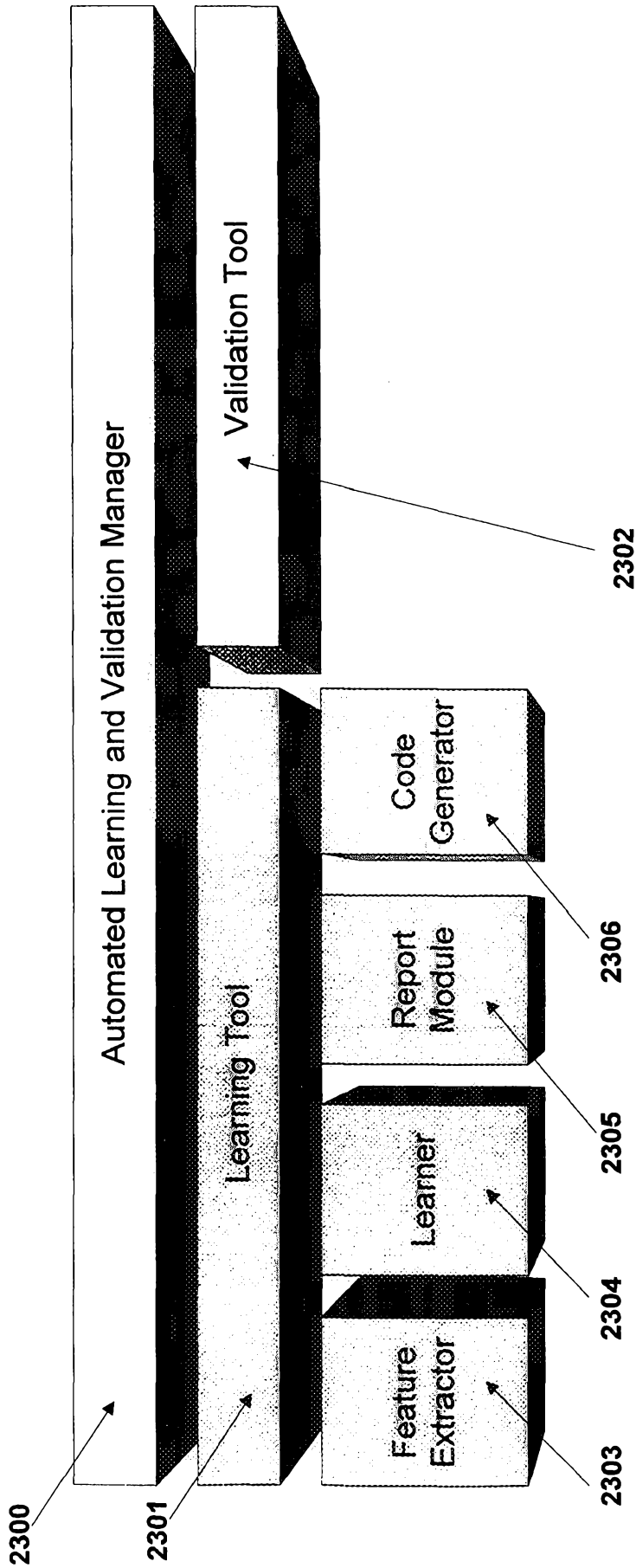


Figure 23

24/30

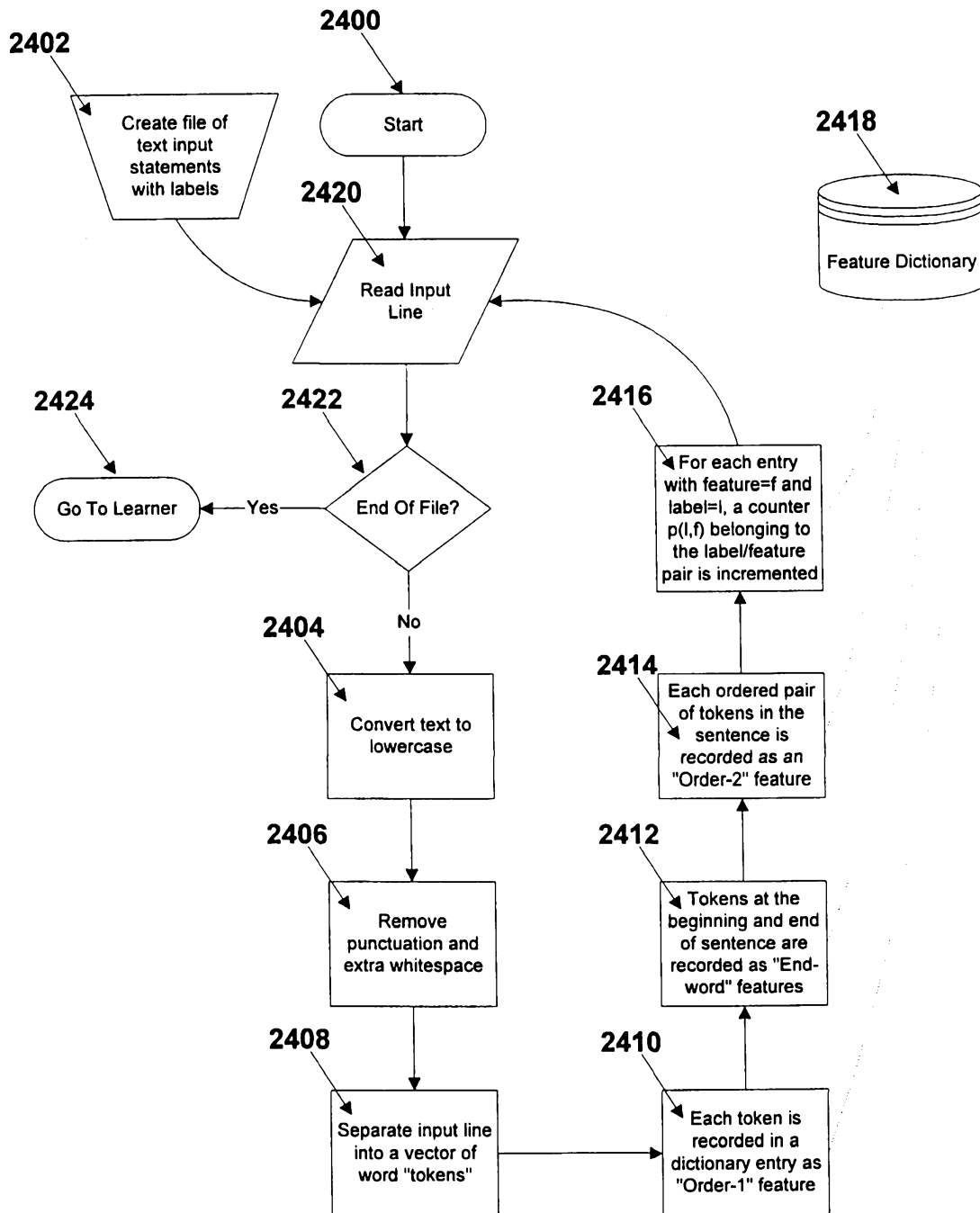


Figure 24

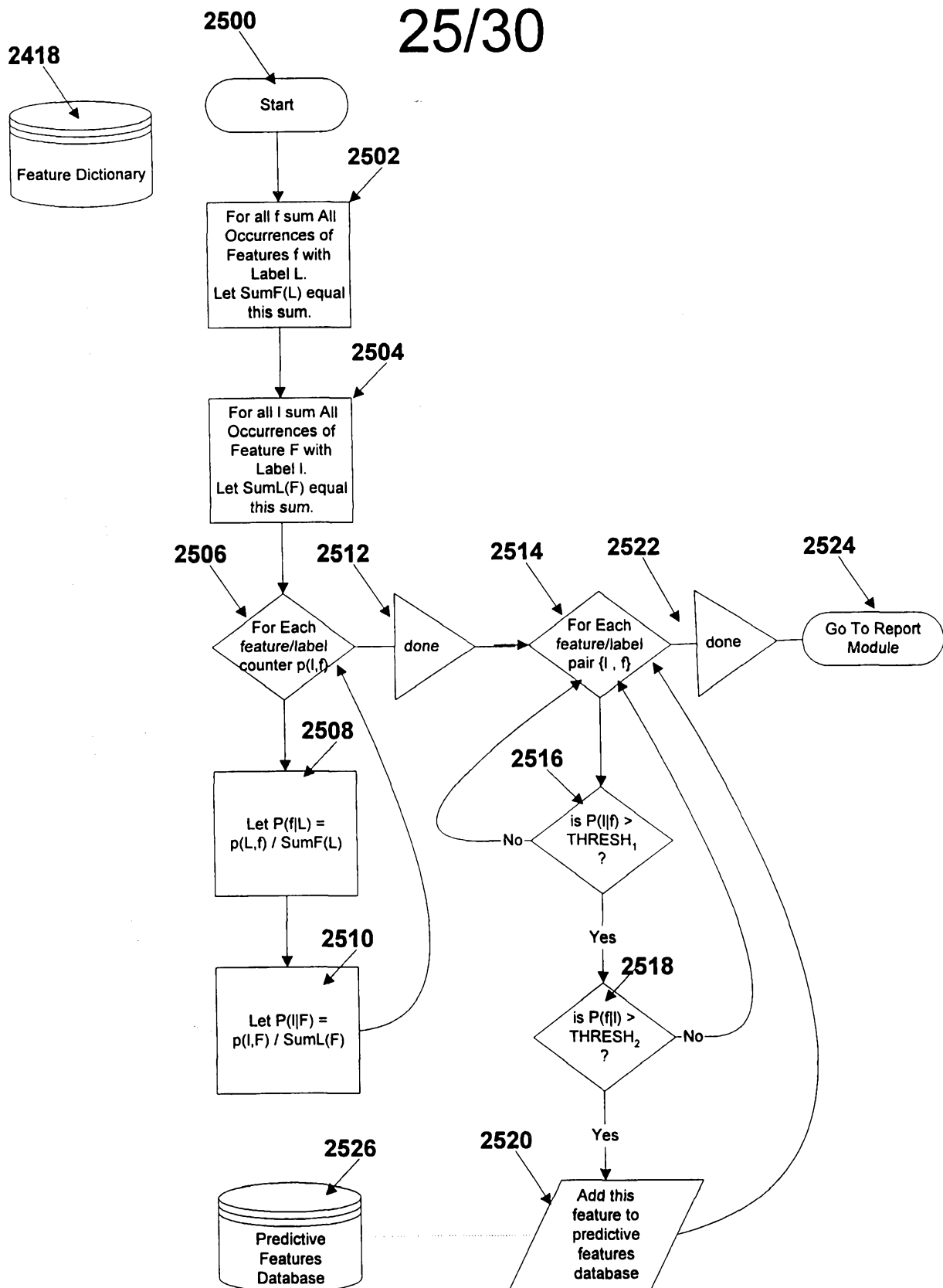


Figure 25

26/30

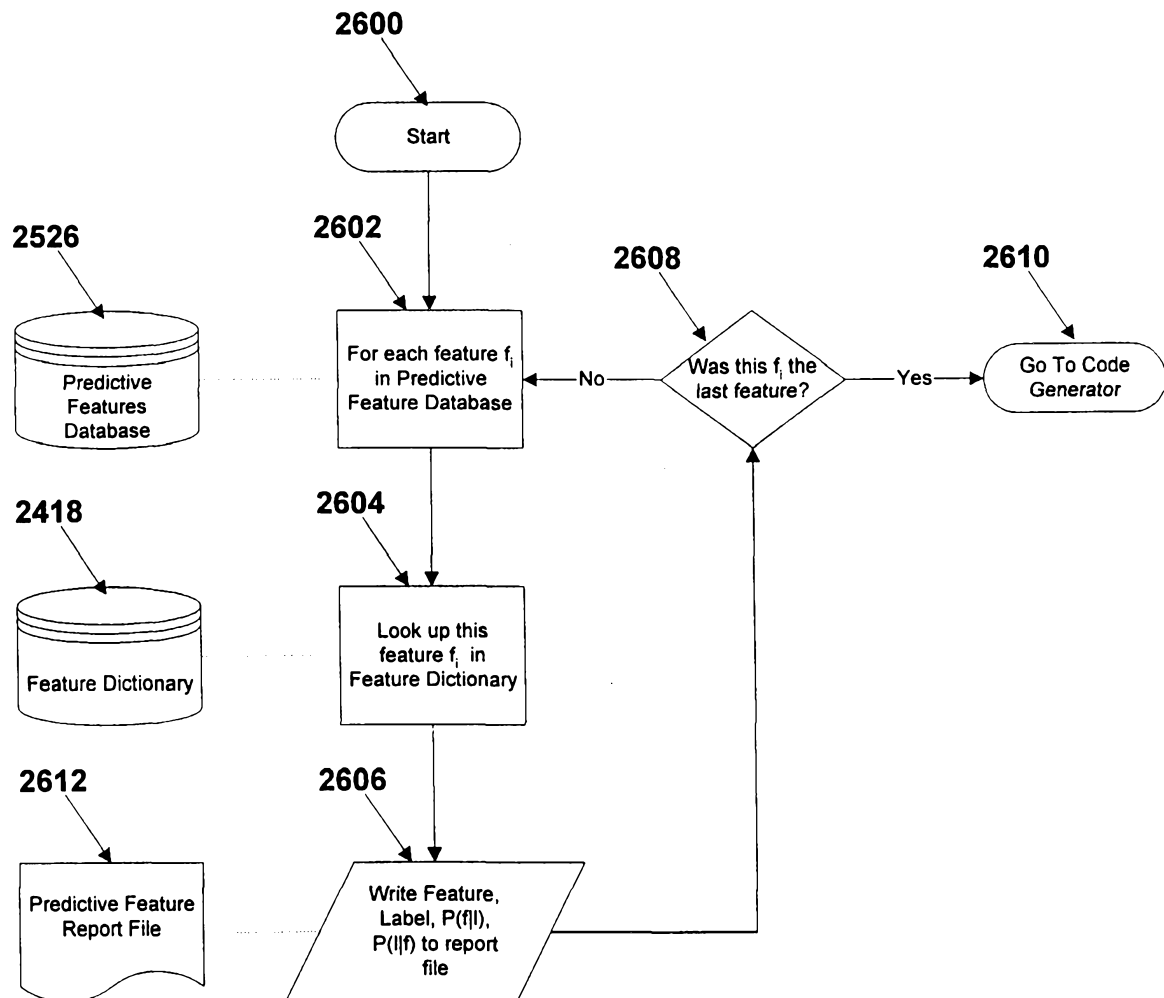


Figure 26

27/30

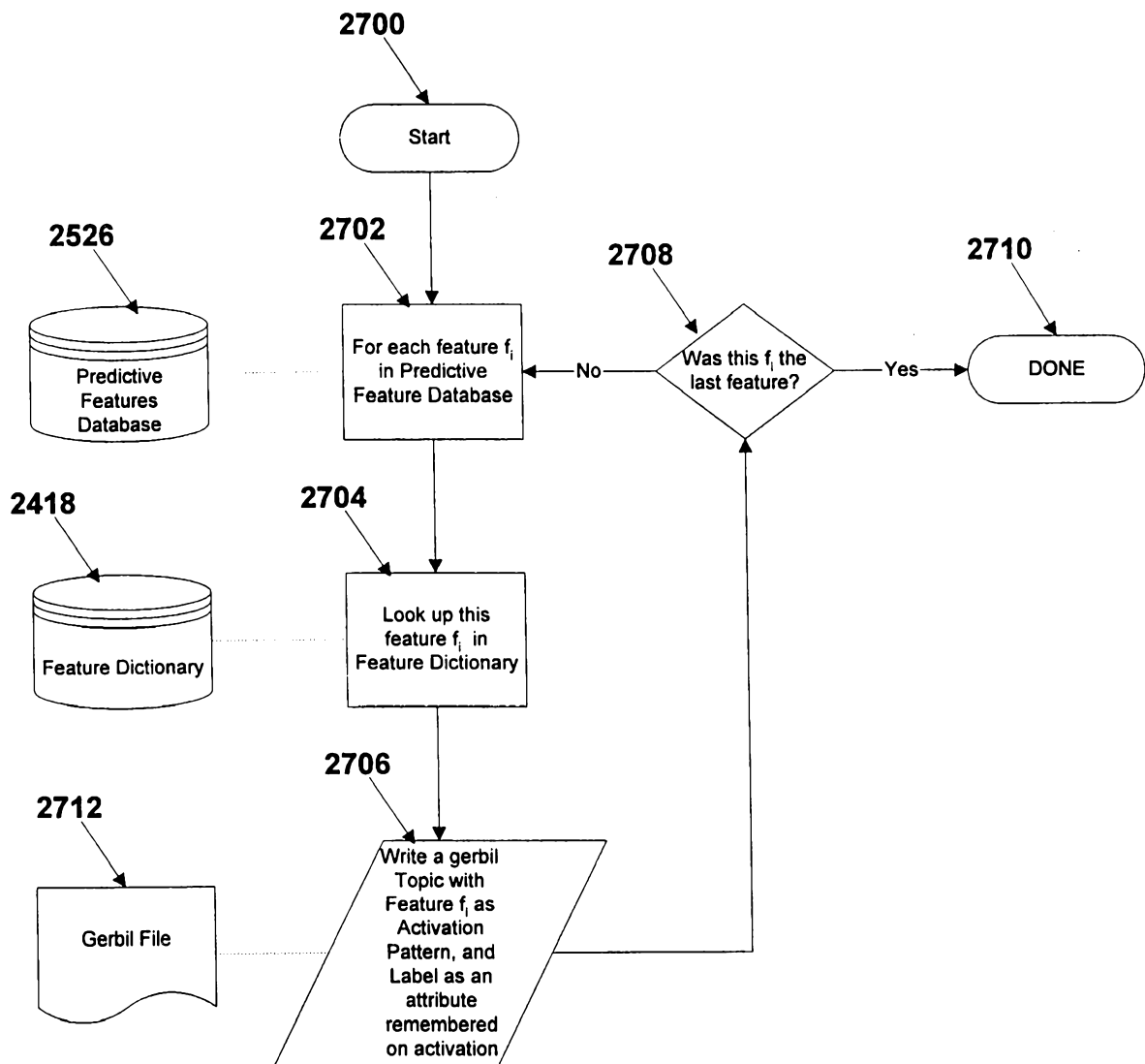


Figure 27

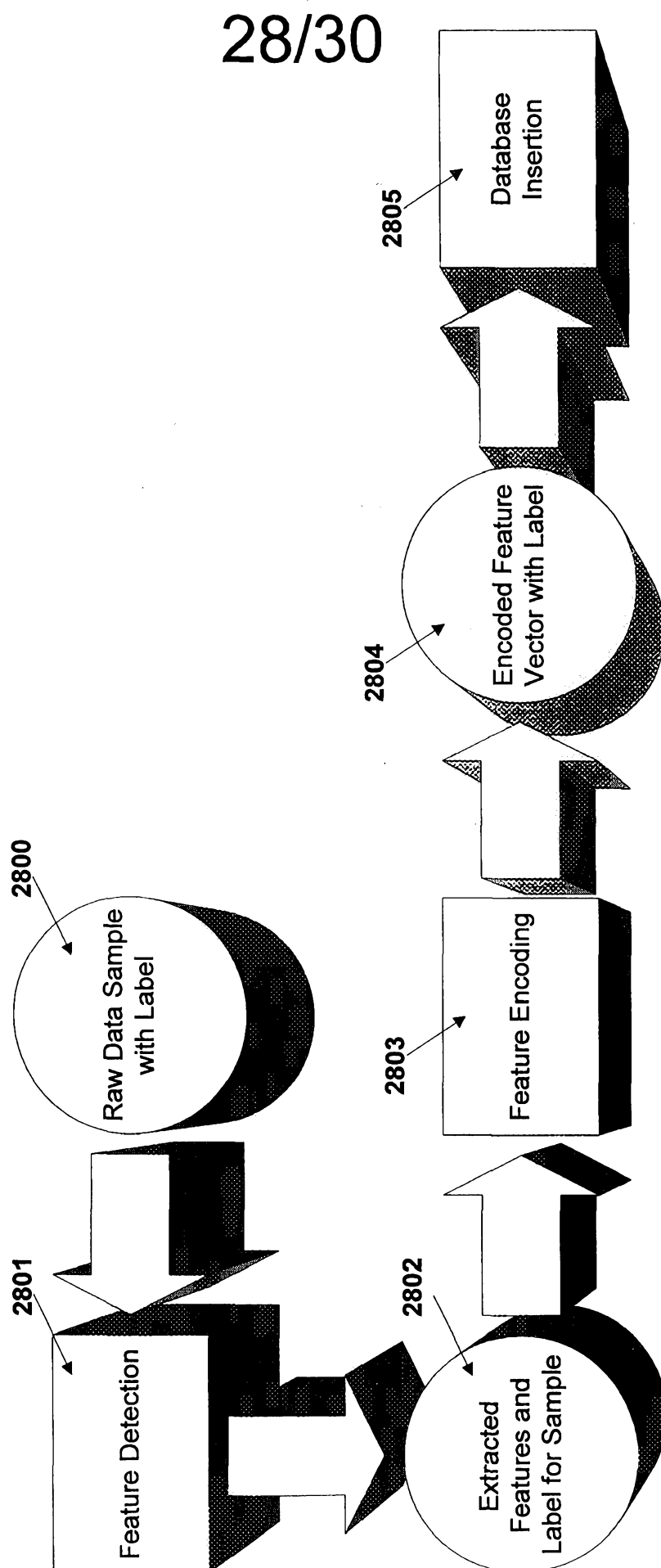


Figure 28

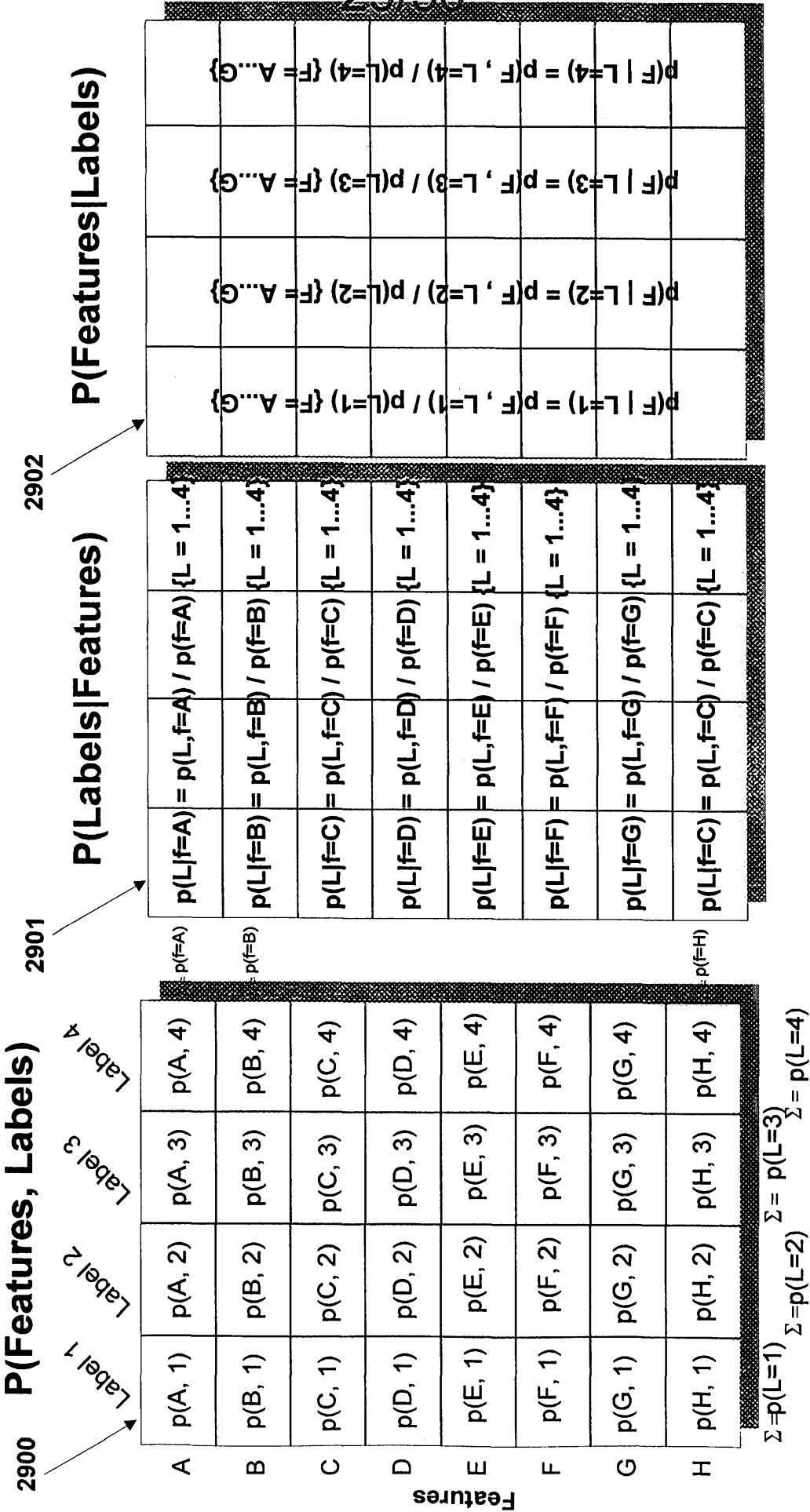


Figure 29

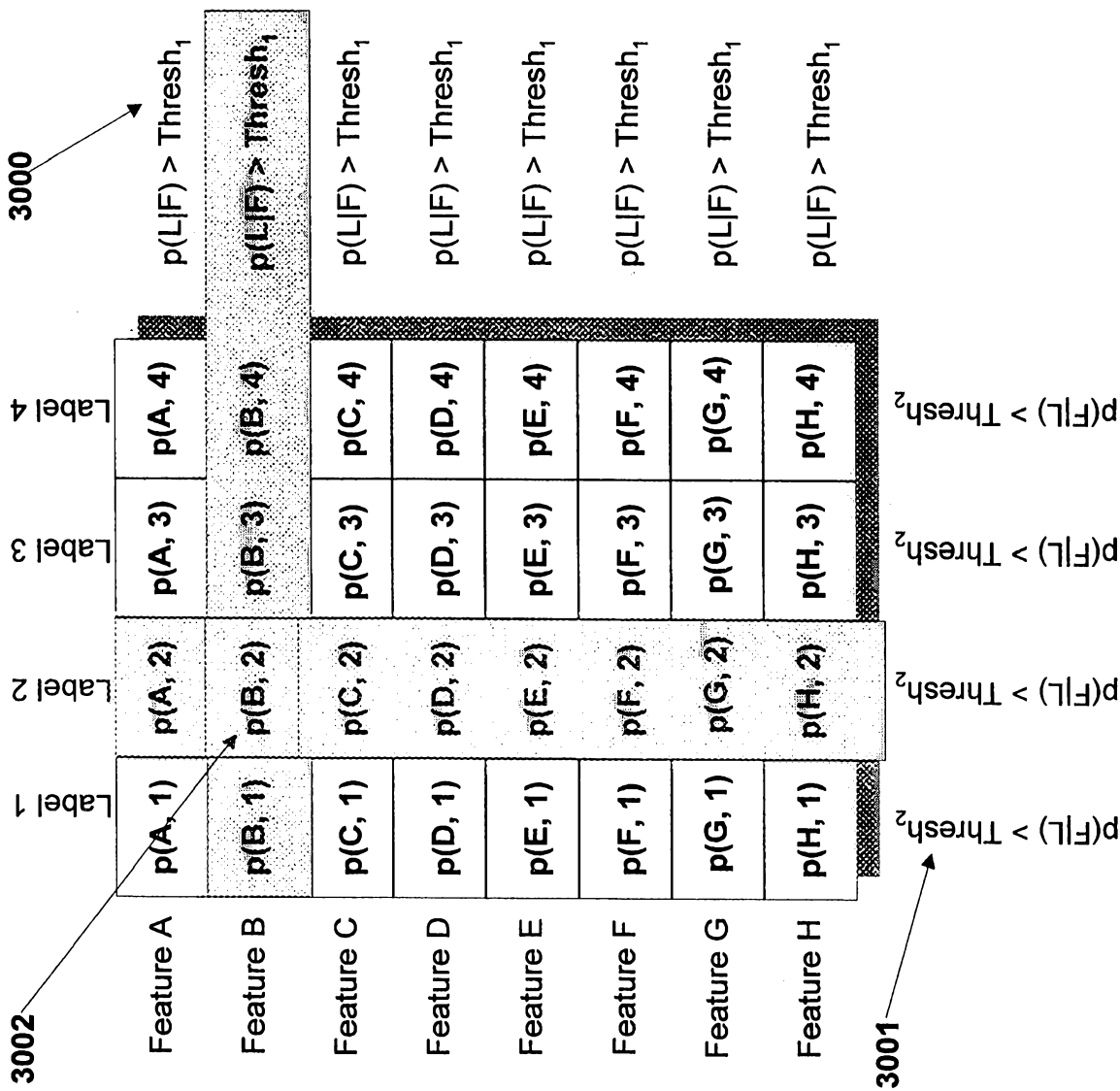


Figure 30

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US98/10536

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G05B 11/00; H04M 1/64

US CL :364/188; 704/270, 277, 278

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 364/146, 188; 379/88; 704/270, 277, 278

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS: (script? and conversation? and virtual and categor?)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5,384,893 A (HUTCHINS) 24 JANUARY 1995, SEE ABSTRACT AND COL. 3, LINE 45 TO COL. 4, LINE 60.	1, 2, 22-24, 44, 45, 60, 61, 66, 86, 104, 106.
Y	US 5,526,407 A (RUSSELL ET AL.) 11 JUNE 1996, SEE COL 6, LINE 40 TO COL. 7, LINE 33; COL. 15, LINES 15-54.	1-8, 22-24, 30, 40-46, 50, 54-66, 68, 70-72, 74-76, 78-81, 85, 86, 98, 104-108, 112, 113.



Further documents are listed in the continuation of Box C.



See patent family annex.

* Special categories of cited documents:	*T* later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
A document defining the general state of the art which is not considered to be of particular relevance	*X* document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
E earlier document published on or after the international filing date	*Y* document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
L document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	*Z* document member of the same patent family
O document referring to an oral disclosure, use, exhibition or other means	
P document published prior to the international filing date but later than the priority date claimed	

Date of the actual completion of the international search

14 OCTOBER 1998

Date of mailing of the international search report

12 NOV 1998

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231

Facsimile No. (703) 305-3230

Authorized officer

Paul P. Gordon

Telephone No. (703) 305-9760

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US98/10536

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y, P	US 5,758,323 A (Case) 26 May 1998, col. 2, line 12 to col. 6, line 65; also see figures 2 and 3.	1-8, 22-24, 30, 40-46, 50, 54-66, 68, 70-72, 74-76, 78-81, 85, 86, 98, 104-108, 112, 113.