EPIC Architecture Principles of Operation

David. E. Kieras

Artificial Intelligence Laboratory

Electrical Engineering and Computer Science Department

University of Michigan

Advanced Technology Laboratory Building

1101 Beal Avenue, Ann Arbor, MI 48109-2110

Phone: (734) 763-6739; Fax: (734) 763-1260

Email: kieras@eecs.umich.edu

Anonymous ftp for documents: ftp://www.eecs.umich.edu/people/kieras

Web home page: http://www.eecs.umich.edu/~kieras/

Purpose of this Document

This document describes how EPIC works from the psychological theoretical perspective, to provide the user of EPIC with information about the psychological assumptions implemented in EPIC. The document does not include details on the software implementation, nor does it present how to build an EPIC model. At this time, this document does not include the justification and citations for how the specific features of EPIC are based on the psychological literature. This document concerns the current version of EPIC, written in C++; for information about the previous LISP version of EPIC see the above website.

This document is subject to revision; the basic structure of the architecture has been fairly stable for some time, but specific details are modified from time to time and new features (such as motor styles) are also added. Thus, please ask about the current state of the architecture before quoting this document.

The first section of the document describes the overall architecture of the simulation software and some key software concepts used in EPIC. The remainder of the document presents the details of the psychological mechanisms in the individual processors and storage systems.

Software Architecture Overview

Conceptual Organization

Conceptually, the EPIC architecture (Figure 1) consists of a set of interconnected processors that operate simultaneously and in parallel. The external environment is represented in the software by a Task Environment Module, or *device*. Although conceptually the external device is not the same kind of thing as the psychological processors, it is simply represented as an additional processor that runs in parallel with the EPIC processors. The device module is programmed in C++, and its programming has no psychological significance except that it represents the external environment and how it interacts with the simulated human. This document currently does not provide any information on how to program the device module, but does describe the interface that connects the device with the simulated human.

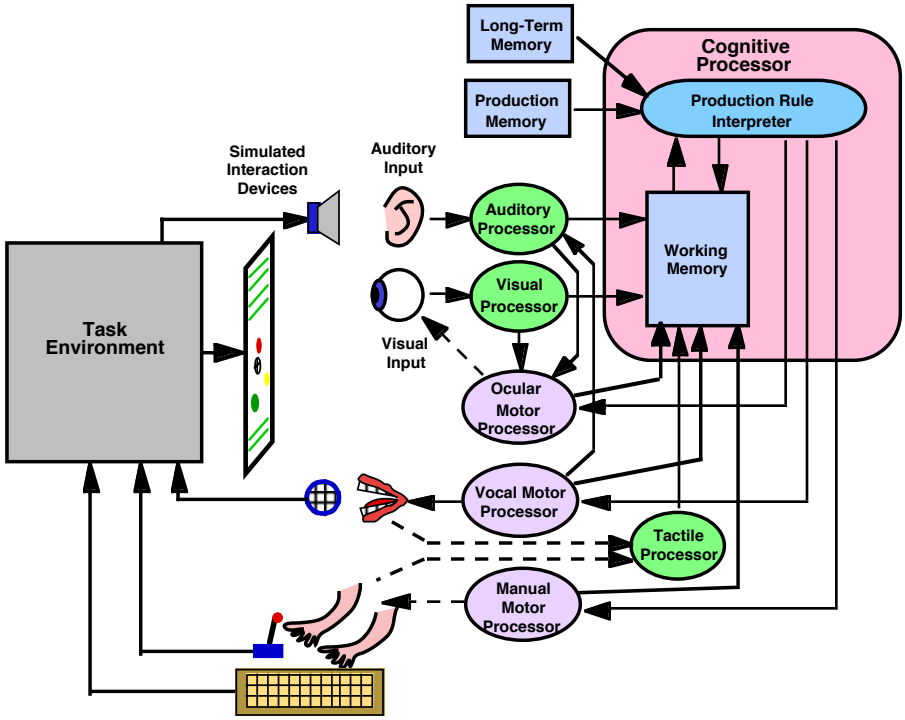

Figure 1. Overall diagram of the EPIC cognitive architecture, showing the task environment module and the EPIC components. The EPIC processors are shown as shaded ovals or boxes. The visual and auditory perceptual processor actually contains a series of processors and stores.

Implementation of parallel processor activity

Some basic concepts of how the EPIC software works are important to understand. The EPIC software simulates the activity of several processors that run simultaneously and in parallel, with information moving among them. The EPIC software uses a simple approach to performing this simulation, but there are certain limitations and pitfalls involved.

The EPIC software uses an *event-driven simulation* approach implemented with Object-Oriented Programming methodology. Each processor is represented as an object instantiated from a class such as Cognitive_processor. The processors respond to Event objects that are sent to them from other processors. There is a single global object, the Coordinator, which has a queue of Event objects. Each Event object (called simply an *event* from here on) contains a pointer to the processor that is supposed to receive the event, a time at which the event is supposed to arrive at its destination processor (in milliseconds), and other data about the event. The event queue is ordered by the time in each event. The Coordinator takes the first item out of the queue, sets the "master clock" that contains the current simulation time to the time of the event, and then sends the event to the destination processor. When that processor has completed its handling of the event, the Coordinator takes the next event out of the schedule queue and repeats the process. Thus the simulation's current time advances not in single increments, but in jumps depending on the arrival time of the individual events.

When a processor receives an event, it can generate output to other processors by creating a new event of the appropriate type that contains a pointer to the destination processor, and giving it to the Coordinator to add to its queue of events (this process is called *sending an event* from here on). Note that a processor can send an event to itself; for example, when the Cognitive_processor gets a "run a cycle" event, it sends itself an new "run a cycle" event scheduled to arrive 50 ms later. Memory processors can send an "remove from memory" event to themselves to cause the loss of an item from memory.

In this scheme for simulating time and parallel processing, the event-handling code in a processor actually takes zero simulation time to run; simulating the effect of non-zero execution time is achieved by the fact that the processor's output events do not arrive at another processor until the simulated future time. Thus, in order to simulate a time delay during a processor's execution, the processor sends an event to itself to arrive in the future; when this event gets handled, the proper simulated time has passed.

Now to complete the basic story at this level, we need to translate the above "OOP speak" to the corresponding C++ concepts. When a processor sends an event to another processor, the following happens: A member function of a processor class will create an event object and supply the pointer to the event to the Coordinator by calling Coordinator::schedule_event with the pointer as an argument. This Coordinator function adds the pointer to the event queue in the proper place in the schedule. When the event pointer comes to the front of the queue, the Coordinator removes it from the queue, and uses it to access the time of the event and the pointer to the destination processor. Using the destination processor pointer, it calls the appropriate event-handling member function of the destination processor class and supplies the event pointer as an argument. When this function returns, the event has been handled, and it can now be deleted, and the Coordinator takes the next event pointer out of the queue.

There is a significant limitation built into this simple and efficient simulation scheme: It commits each processor to being "ballistic" in the sense that there is currently no way for a processor's output for a previous input to be modified by a subsequent input. For example, suppose the auditory perceptual processor receives sensory input at simulated time 1000, and in response, instantly generates an output to the cognitive processor to arrive 200 ms later, at simulated time 1200. Then another stimulus arrives at time 1100, when in psychological real time, the processing of the first stimulus is still underway. But in simulated time, the processor has already produced the output to the first stimulus, and so it cannot be modified as a result of the second stimulus. Thus, in the current version there is no way in which the auditory processor can "take back" its previously-generated output and replace it with a different one. While this ballistic character might appear to be restrictive, thus far it has not presented a serious problem with implementing the architecture. The basic workaround to the example problem is to arrange each processor to simulate time delays via self-scheduling to ensure that events happen in the proper sequential order in simulated time. That is, if there is a need to "take back" an output on occasion, instead of dispatching the output ballistically, the auditory processor can send a "send it now" event to itself to arrive at a later time. When the "send it now" event arrives, the processor can then send the output to the next processor immediately. But first, it can check to see if a "canceling" event has been received in the meantime, and if so, simply discard the "send it now" event.

Finally, in the current implementation, the perceptual processing is "change driven" in the sense that the information passed into and between processors is about *changes* in state of perceptual objects, rather than about the current state. For example, if a visually presented object appears, the processors and stores in the visual system send and receive Visual_appear_events. If the object changes location, then a Visual_Change_Location_event is propagated through the system. If a processor needs to know the current state of affairs, it must maintain its own record based on the history of changes. The current software has a couple of awkward aspects, e.g. in the modeling of saccadic suppression, due to the need to propagate change information in order to maintain state information.

Symbols

A small note: The EPIC software makes heavy use of Symbols, which is a flexible data object that can contain a string of characters, a single numeric value, or a pair of numeric values. The numeric values are represented as double-precision floating point numbers. Thus a Symbol can carry the most common kinds of values. Symbols can be compared for equality and ordering. If both Symbols contain numbers, or both contain strings, the normal comparisons are made. However, if a numeric value is compared to a string, the numeric value is always smaller or unequal. The C++ code implementing the architecture uses Symbols heavily, but at the modeling level, this aspect of the code is important only for implementing the simulated device.

Processing Time Parameters

Each processor has a set of time parameters that determine how long the processing takes, expressed in integer milliseconds. These parameters can be set and controlled by specifications in the Production Rule Set (.prs) input

files. The parameters are all independent of each other; they can be set to fixed values, or values that are randomized in various ways. Each parameter has a name that must be unique compared to the names of other parameters for the processor, but can be duplicated between processors. By default, the parameters are set to standard or typical values in the compiled simulation code. These default values and settings can be overwritten by specifications in the Production Rule Set (.prs) input files. The values in effect are displayed when the simulation is started up. The possibilities and how to modify them can be inferred from the dump of all parameter values at the start of a run.

Parameter Specification Syntax

The parameters have values and settings that can be altered from the production system file using the following syntax:

(Define Parameters <parameter_specification> ... **)**

A <parameter_specification> consists of:

(<processor_name> <parameter_name> <specifications>**)**

The specifications depend on the processor and parameter name, and come in two varieties, simple and complex.

Simple parameters

A simple parameter specification is a single (possibly random) value for a parameter. A simple parameter specification has the form:

(<processor_name> <parameter_name> <randomization_specifications> <values> **)**

More precisely, <randomization_specifications> and <values> have the following possibilities:

If the <randomization_specifications> is Fixed, then <values> is a single value. For example:

(Eye Eccentricity_fluctuation_factor Fixed 1.0)

Otherwise, <randomization_specifications> consists of the name of distribution followed by when the distribution should be sampled.

Possible distributions: Uniform, Normal

If the distribution is Normal, <values> must be a mean and standard deviation.

If the distribution is Uniform, <values> must supply a mean value, followed by an optional deviation that gives the range above or below the mean; if not supplied, the deviation is assumed to be the mean/3, which for a Uniform distribution, will produce a coefficient of variation of 20%.

Possible sampling times: When_used, On_command

The value of a parameter is always its mean value unless it is sampled at some time.

When_used means that every time the parameter value is accessed (used) by the EPIC software, a new value is sampled.

On_command is present to allow a possible future implementation in which e.g. the experimental procedure (performed by the device) could call for a new randomization at the start of a trial; the last sampled value is used in the meantime.

Since overwhelmingly, parameters specify processor time values, the values of a sample parameter are required to be positive; if a zero or negative value is sampled from the specified distribution, a new value is sampled until a positive one is obtained.

Example:

(Eye Eccentricity_fluctuation_factor Normal When_used 1.0 .1)

The Eye processor has a parameter named "Eccentricity_fluctuation_factor". It is Normally distributed, and will be sampled every time the parameter value is used by the Eye processor; the mean is 1.0, and the standard deviation is 0.1.

Complex parameters

A complex parameter is a parameter whose value depends on another symbolic value, such as the time to recode a particular visual feature. Such a parameter is might be only a single fixed value, or might be the name of a complex function rather than a value. These tend to be processor-specific in syntax, and are described with the relevant processors. For present purposes, in the following example, the Recoding_time parameter for the Visual_perceptual_processor has a value of 50 ms for the visual property of Text:

(Visual_perceptual_processor Recoding_time Text 50)

The Cognitive System

Cognitive_processor

The cognitive processor, is described first because the output of the perceptual processors must be explained in terms of the cognitive processor's production system mechanism, which is implemented using the Parsimonious Production System (PPS) production rule interpreter. Production systems have a memory, which is usually called "working memory" in the context of production rule interpreters, but here it is called the *production rule memory* to distinguish between the data structures required to match against the production rule conditions (the *production rule memory*) and various psychological concepts about working memory, which are not necessarily synonymous with the production rule condition states. The memory systems for Long-Term Memory, and Production Memory, are shown in the diagrams as separate boxes connected to the cognitive processor. This is psychologically meaningful, but in misleading in terms of the actual software organization. That is, the contents of Long-Term Memory are simply items in the initial state of the production rule memory that are neither added nor removed during task execution. Production Memory is simply the set of production rules in the model.

The cognitive processor receives inputs from the perceptual processors and state information from the motor processors. It operates on a cyclic basis, running every Cycle_time, normally 50 ms. Each cycle consists of updating the production system production rule memory using the results of the previous cycle, matching the production rules against the current contents of the production rule memory, and then executing the rule actions for all rules whose condition matched the contents of the production rule memory. The actions consist of either modifications to the production rule memory to be made at the beginning of the next cycle, or sending commands to the motor processors.

The production rule memory consists of several types of items, which in PPS are distinguished only as a matter of convention and output format. Production rule memory contains *clauses*, which are items in the form of a list; the first symbol in the item is the "type" of the clause. PPS prints the current contents of the production rule memory grouped by the types. There is no other effect of the clause type. Currently, as presented in Kieras et al. (1999), in EPIC these types are Goal, Step, Strategy, and Status for flow-of-control items, and Visual, Auditory, Tactile, and Motor for the *modality-specific partitions* of working memory, and LTM for long-term memory. A special type, Tag, is used for a special form of working memory - "tagging" perceptual objects with their roles in production rules (e.g. (Tag Visobj23 Current_stimulus), and the type of WM is used (with reluctance) as for clauses assumed to be in an unspecified psychological *amodal working memory*. Note that a special feature of PPS is that these types and partitions are not "hard wired" into the PPS interpreter, but are defined only by what clause items the production rules happen to use; thus the production system interpreter does not have to be altered to reflect changing hypotheses about the different kinds of working memory; rather the hypotheses are represented by a consistent pattern of reference to production rule memory items in the production rules.

The final product of the perceptual systems are updates to the PPS production rule memory, normally consisting of a clause that is to be deleted and a clause that is to be added. For example, if visual object #23 was green, and it changes to red, the clause (Visual Visobj23 Color Green) will be deleted, and (Visual Visobj23 Color Red) will then be added. These changes to the production system memory are made immediately by the perceptual systems. Because the perceptual processors are asynchronous "pipelines" and the cognitive processor is intermittent and cyclic, the perceptual processors can produce output that potentially reports a change from one perceptual state to another within a single cognitive cycle. In contrast, if a rule specifies that a particular clause is to be deleted and another added, these changes are accumulated in a list which is processed at the beginning of the production rule cycle; all the deletions are done first, followed by all the additions.

Likewise, the motor processors inform the Cognitive_processor of their state changes by likewise posting immediate updates to the production system memory.

Event handling:

Cognitive_Update_event

Updates from the perceptual and motor systems are applied to the production system memory immediately.

Cognitive_Cycle_event

1. Update the production system memory using the results from the previous cycle:
   Delete all clauses in the delete list from the previous cycle.
   Add all clauses in the add list from the previous cycle.
   Clear the delete and add lists.
2. Fire any and all rules whose conditions are currently true.

Add and Delete actions put clauses in to the add and delete lists for the next cycle.
Any output to other processors is sent at current time + Cycle_time - 1. The -1 ensures that motor processors will commence executing commands prior to the next cognitive processor cycle.

3. Schedule another Cognitive_Cycle_event to arrive at current time + Cycle_time.

Current parameter values

Cycle_time: 50 ms

The Production System

Production system implementation

The cognitive processor uses a version of the Parsimonious Production System (PPS) production rule interpreter, described earlier in Covrigaru & Kieras (1987) and in Bovair, Kieras, & Polson (1990). PPS provides a simple and flexible production rule implementation well suited for task performance modeling. Its main benefit are that the production rules are easy to write, simple to read, and have behavior that is easy to understand and to predict. PPS use a rete-match algorithm based on Forgy(1982) for speed, but no claims are made for its speed over its ease of use. In this algorithm, the set of production rules is compiled into a matching network; each node in the matching network represents a partial match to a set of production rule conditions. Then during execution, update clauses are processed through the matching network to simultaneously update the state of every rule in the production system.

Notation Conventions. The LISP EPIC version of PPS followed the ancient LISP tradition of using all upper-case letters. The current PPS and EPIC in general uses mixed-case notation, with a convention that reserved and predefined terms begin with an upper-case letter. To allow backward compatibility, an effort has been made to recognize all-upper-case reserved words in production rules, but users should try to use the mixed case versions instead. Since there are other small differences in the details of the architecture, most models written for the LISP version will require some modification to run in the current version.

Production rule syntax

The PPS production rule memory consists of a set of *clauses*, which are simply lists of constant terms which can be taken as assertions for the truth of some proposition. For example, by convention, the clause (Visual Visobj23 Color Green) means that the color of a certain visual object is green. PPS works in terms of cycles; at the beginning of each cycle, the production rule memory is updated. Then the conditions of the production rules are matched against the whole set of clauses. If the condition specified by a rule has a matching set of clauses in the production rule memory, then the rule is said to *fire*, and the actions of the rule will be executed. These actions typically include adding or removing clauses from the production rule memory, thereby changing which rules might fire next. But note that the changes to the production rule memory are queued up for the beginning of the next cycle. PPS can fire multiple rules on each cycle; the rules are fired effectively in parallel since no changes to the production rule memory are made until the beginning of the next cycle.

A PPS rule has the form:

(<rule-name> **If** <condition> **Then** <action>)

Each rule must have a unique name. The condition is a list of clause patterns, described in more detail below. All of the patterns must be matched for the rule to fire. The action is a list of actions. The syntax for conditions and actions is described below.

Production rule conditions

The condition of a production rule is a list of clause patterns:

(<clause pattern> <clause pattern> ...)

and each clause pattern is simply a list of *terms*, which are *symbols*:

(term term ...)

If there is more than one clause pattern in a condition, they are treated as a conjunction; all of the clause patterns must be matched before the rule can fire.

In addition, a clause pattern can be negated:

(**Not** <clause pattern>)

Finally, a negation can include multiple clause patterns, which are treated as a negation of the conjunction:

(**Not** <clause pattern> <clause pattern> ...)

At present, this pattern behaves identically to a series of individual (Not ...) clause pattern, which will probably be changed in the future to a more useful definition.

Each term in a clause pattern can be a constant symbol, a matching variable indicated by a prefix of "**?**", or a wildcard term of "**???**". The number and order of the terms is significant. For a match of an *individual* clause pattern to occur,

there has to be at least one clause in the production rule memory such that:

The clause pattern has the same number of terms as the production rule memory clause.

Each constant term in the clause pattern is the same as the term in the same position in the production rule memory clause.

Each wildcard term in the clause pattern can correspond to any term in the same position in the production rule memory clause.

Each variable term in the clause pattern can correspond to any term in the same position in the production rule memory clause, and the variable will be assigned that term as a possible value.

For the condition of the production rule to be satisfied, all of the constituent clause patterns must be matched, and it must be true that there is a least one set of values for the variables such that all of the patterns containing variables are satisfied simultaneously. A variable and its value is called a *binding*; a set of variable values that match all of the conditions in a rule is termed a *binding set*. Thus a rule whose conditions include variables must have at least one binding set in order to be fired.

For example, consider the following fragment of a rule that finds a superset of two specific things:

(Rule1 If ((ISA COW ?X)(ISA DOG ?X)) ...) ...)

The condition of this rule is matched if the production rule memory contains the two clauses (ISA COW MAMMAL) and (ISA DOG MAMMAL); the binding set can be written as (?X MAMMAL). But if the production rule memory contains only (ISA COW MAMMAL) (ISA DOG PET), the rule condition cannot be matched. That is, the first clause pattern is matched if the variable ?X is assigned the value MAMMAL, and the second clause pattern matches if the same variable, ?X, is assigned the value PET. But since there is no assignment of ?X that satisfies both clause patterns, the condition of this rule is not matched - there are no binding sets.

A more elaborate situation involves negation:

(Rule2 If ((ISA ?X ?SUPERSET) (Not (ISA ?X MAMMAL)) ...) ...

This rule condition will be matched if the production rule memory contains a clause that matches the first pattern but in which the first term identified by the value of ?X is not a MAMMAL. This would be matched if the production rule memory contained (ISA PERCH FISH) and did not assert (ISA PERCH MAMMAL), which should be the case! But the production rule memory clauses (ISA DOG PET) and (ISA DOG MAMMAL) would not result in ?X being assigned DOG as a possible value, because the negated pattern would be matched with the same value of ?X. Thus this rule condition will match for all ?Xs that are not MAMMALs.

Note that the bindings assigned to variables are maintained only within the scope of the matching and action execution of a single rule. Thus if one rule matches with ?X = AARDVARK, another rule that has ?X in its condition is unaffected; ?X may be assigned either the same or entirely different values in the context of the other rule. In other words, binding sets exist only within the evaluation and execution of a single rule.

A caveat concerning variables inside negations. A common PPS programming error is to have a variable inside a negated clause that only appears in that one clause, and then attempt to make use of that variable's binding. For example, if ?Z does not appear anywhere else in the rule condition, (Not (ISA ?Z MAMMAL)) is effectively equivalent to (Not (ISA ??? MAMMAL)) . The condition should be written with a wildcard instead of a variable to avoid confusion.

Finally, and very importantly, if there is more than one set of variable value assignments that satisfy the condition, the rule is multiply instantiated. Thus a multiply instantiated rule is one that fires with more than one binding set, and each such binding set specifies one value for each variable. When the actions are executed, they will be executed once for each binding set. Thus, if the above rule fragment is satisfied with ?X being PERCH, OCTOPUS, or DRAGONFLY, the binding sets will be ((?X PERCH ?SUPERSET FISH)(?X OCTOPUS ?SUPERSET MOLLUSC)(?X DRAGONFLY ?SUPERSET INSECT). The actions will be executed three times, once for each of the binding sets, which are each pairs of values of ?X and ?SUPERSET. This multiple execution will occur within a single rule cycle.

To allow flexibility in coding rules that deal with the multiple instantiation of rules, it is also possible to include in the rule conditions one or more predicates that operate on specified variables. These evaluate to true only if some condition is true of the variables in a binding set, either within or between binding sets. This evaluation is repeated for each binding set in a multiple instantiated rule firing. For example, (Different ?X ?Y) is true only in the case that ?X and ?Y are assigned to different values; if there are multiple possible values of ?X and ?Y, this predicate can be used

to filter out those in which ?X and ?Y happen to have the same values, leaving only the binding sets in which they have different values. Thus the predicates can be viewed as filters that only let pass variable binding sets that meet the condition of the predicate. Regardless of where predicates appear in a rule, they are collected and applied in the order they appeared to any binding sets computed from the rest of the condition. Currently these predicates and their effects are:

(Unique <var1> <var2>) is true for binding sets that have unique values for the variables; that is ones in which var1 and var2 have different values over the entire set of binding sets. For example, consider three binding sets as follows:

{?X = a, ?Y = b}, {?X = b, ?Y = a}, {?X = c, ?Y = d}

The predicate (Unique ?X ?Y) is true only for the first and third sets because in the second set, ?Y has the same value as ?X does in the first set.

(Different <var1> | <constant1> <var2> | <constant2>**)** is true for binding sets in which the first variable or constant is different from the second variable or constant. For example, (Different ?X ?Z) is true of only the first of the following binding sets:

{?X = a, ?Y = b, ?Z = c}, {?X = b, ?Y = a, ?Z = b}, {?X = c, ?Y = d, ?Z = c}

While (Different ?X b) is true of the first and third.

(Equal <var1> | <constant1> <var2> | <constant2>**)** is the opposite of Different; it is true of binding sets in which the first variable or constant is equal to the second.

(Greater_than <var1> | <constant1> <var2> | <constant2>**)** is true of binding sets in which the first variable or constant is arithmetically greater than the second. For example, (Greater_than ?X 42) would be true of all binding sets in which ?X was a numerical value greater than 42.

(Greater_than_or_equal_to <var1> | <constant1> <var2> | <constant2>**)** is true of binding sets in which the first variable or constant is arithmetically greater than or equal to the second.

(Less_than <var1> | <constant1> <var2> | <constant2>**)** is true of binding sets in which the first variable or constant is arithmetically less than the second.

(Less_than_or_equal_to <var1> | <constant1> <var2> | <constant2>**)** is true of binding sets in which the first variable or constant is arithmetically less than or equal to the second.

(If_only_one) takes no arguments, and is true only if all of the variables take on only one value in the entire set of binding sets. This predicate can be used to identify when a rule condition or variable has exactly one match.

(Use_only_one) takes no arguments, and passes through only the first set of bindings. It is used to pick out a single binding set for a multiply instantiated rule.

(Randomly_choose_one) takes no arguments, and is like Use_only_one, except it picks a set of bindings at random. For example, it can be used to pick an object to process at random.

Example of predicate usage

An example of one of the more important uses of these predicates starts with the following rule fragment:

(Rule2 If ((ISA ?X ?SUPERSET)(ISA ?Y ?SUPERSET)) ...

This rule condition can be matched by a single clause in the production rule memory, e.g. (ISA NEMATODE WORM) simply by assigning both ?X and ?Y the value NEMATODE, and ?SUPERSET the value WORM. This matches in PPS because PPS does not include any "data refractoriness" that restricts a production rule memory clause to matching only a single clause pattern. Adding (Different ?X ?Y) to the condition means that the rule will fire only if there are two distinct values for ?X and ?Y. Thus the set of bindings for ?X and ?Y will correspond to all pairs of items that share the same superset. However, note that each such pair will get represented twice, since the match will include one assignment to?X and the other to ?Y, and vice versa. For example, these two binding sets could be produced, and the actions executed for both, even though they are probably redundant to the function of the rule:

{?X = DOG, ?Y = CAT, ?SUPERSET = MAMMAL}
{?X = CAT, ?Y = DOG, ?SUPERSET = MAMMAL}

Adding (Unique ?X ?Y) to the condition will filter out these cases in which ?X and ?Y take on the same value. Finally, adding (Different ?SUPERSET MAMMAL) will filter out the matches in which the superset was

MAMMAL, leaving the following final example condition, which will match only distinct, different pairs of items that had non-MAMMAL supersets:

(Rule2 If ((ISA ?X ?SUPERSET)(ISA ?Y ?SUPERSET)(Different ?X ?Y)(Unique ?X ?Y)(Different ?SUPERSET MAMMAL)) ...

Production rule actions

The action of a production rule consists of a list of function-like invocations. These are executed in order when the rule is fired. Two essential actions are:

(Add <clause to be added to the production rule memory>**)**

(Delete <clause to be deleted from the production rule memory>**)**

The clause to be added or deleted is a list of terms, as described above. However, each term in the clause is examined; if it is a production rule condition variable (defined with a prefix of "?"), it is replaced with the constant that is the value of the variable in the current binding set. For example, suppose the production rule memory contains the clauses (ISA FIDO PET) and (COLOR FIDO BROWN); the following rule will remove (COLOR FIDO BROWN) and add (FIDO IS BROWN IN COLOR) to the production rule memory:

(Rule3 If
((ISA ?X PET)(COLOR ?X ?Y))
Then
((Delete (COLOR ?X ?Y))(Add (?X IS ?Y IN COLOR)))

As described above, it is possible for a rule to be fired with multiple binding sets. If this occurs, then the actions will be executed once for each binding set. Thus in the above example, if the production rule memory also had (ISA PETUNIA PET) and (COLOR PETUNIA BLACK), then in one production system cycle, this rule would remove both (COLOR ...) clauses and add both the clause (FIDO IS BROWN ...) and (PETUNIA IS BLACK ...). As mentioned above, the scope of the binding sets is limited to the individual production rule firing. Thus, the value FIDO of ?X in this example rule is not available to any other production rule or any other cycle; effectively, the variables are bound to their values in the condition of the rule, and retain the values during execution of the action of the rule, and then the values are "erased" and no longer available. Thus to pass information rule-to-rule or cycle-to-cycle, the information must be explicitly stored in the production rule memory and matched by rule conditions, rather than being moved invisibly or tacitly through variables.

EPIC includes a basic action that allows PPS rules to control motor processors:

(Send_to_motor <modality> <command> <specifications>**)**

The command and its specifications are sent to the specified motor processor, and arrive at the beginning of the next cycle. (The actual time of arrival is current time plus the current cognitive processor cycle time minus 1 ms). The specific commands and specifications depend on the motor processor, and so are described there.

Finally, as an aid to debugging, PPS has a logging action:

(Log term term ....**)**

If the rule is executed , a time-stamped log entry is written to the normal output stream; the body of the entry is the terms in the order listed, which can consist of symbols, production rule variables (which are replaced with their values), and string literals. As many terms can be included as desired.

The Visual System

Visual Representation and Processing

Visual Processing Overview

Figure 2 shows the detailed structure of the visual processing systems. This includes both the visual input systems and the ocular motor processors, to be discussed later.

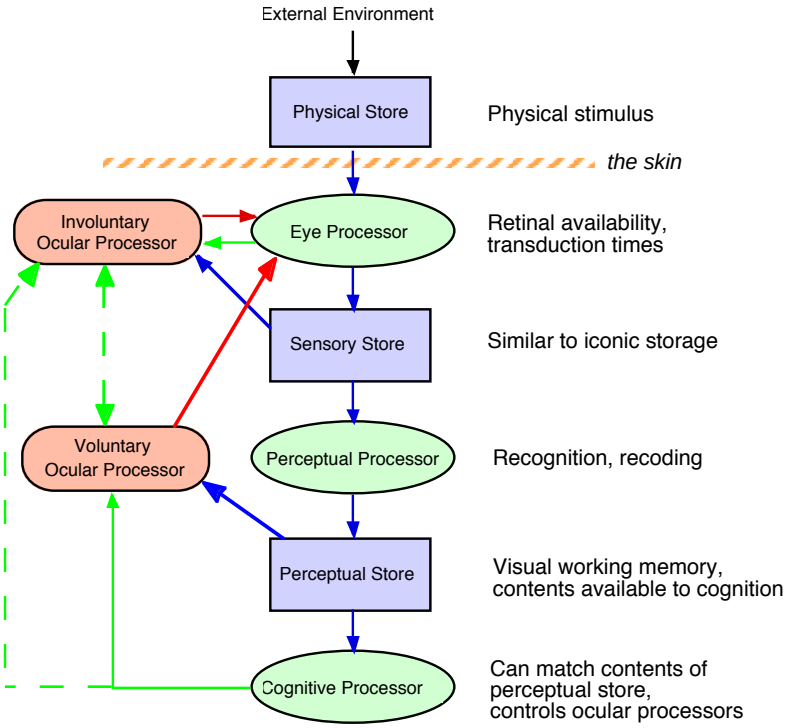

Figure 2. The EPIC visual system: A chain of stores and processors on the input side, and two ocular motor processors to control the eye's orientation. The External Environment is the Device. Blue lines show the flow of information about visual input; green lines show command control relationships, and red lines show muscular control relationships.

By design, this is psychologically a very traditional representation, and the software structure corresponds quite directly to this representation. In summary, a *visual object* is a named collection of property-value pairs. The Device_processor can add or modify visual objects in a human's physical environment by calling Human::make_visual_object_appear and Human::set_visual_object_property, with the *physical name* of the visual object. This results in an object being added or modified in the Visual_physical_store for the human; this store represents in effect the light pattern currently falling on the human's outer surface. Depending on the current orientation of the eye, the Eye_processor will pass visible events through to the Visual_sensory_store (following a transduction delay), where a new representation of the object is constructed, this being a psychological object that has a *psychological name* assigned by the Eye_processor. When an object is added or modified in Visual_sensory_store, the Visual_perceptual_processor is immediately notified. It can use any information present in the Visual_sensory_store to decide how to encode the perceptual properties of an object, and send the results as events (with encoding delays) to the Visual_perceptual_store. This final store corresponds to what is usually termed Visual working memory; appearance and changes to objects in Visual_perceptual_store become updates to Cognitive_processor's production system memory as described; at all times, the contents of Visual_perceptual_store and the production system memory are supposed to be in exact synchrony, meaning that in terms of psychological theory, the Visual partition of the production system memory is supposed to be synonymous with the the visual working memory in Visual_perceptual_store. Objects and their properties are retained in Visual_perceptual_store as long as they are visible; once they become invisible (either through eye movements or disappearance) the object or its properties are erased from Visual_perceptual_store after a time.

The involuntary ocular processor, Invol_ocular_processor, is informed by Eye_processor of certain changes in visual objects. The appearance of an object, or a change in location of a foveated object, can cause an

involuntary eye movement. Invol_ocular_processor makes use of information in the sensory store

Symbolic representation

Because of its focus on overall task performance, the EPIC architecture includes no mechanisms for performing the actual pattern recognition involved in vision. Thus visual inputs are symbolically coded visual properties, rather than descriptions of stimuli at the physical level. In the software, a visual object is represented as an arbitrary symbol with a list of property-value pairs that specify the visual attributes of the object.

Visual Space Coordinates

Visual space contains objects in two-dimensional space, with locations whose coordinates are of the center of the object in terms of visual angle (in degrees), with the origin representing "straight ahead of the human's nose." Currently there is no representation of depth (i.e. distance from the human's nose) or of head position. The sizes of objects are also represented in degrees of visual angle of the sides of a rectangular bounding box centered at the object location and enclosing the object.

Visual Objects

A *visual object* is a named collection of property-value pairs. Two of the properties of a visual object are separately represented and must always be present: these mandatory properties are the geometric location and size of the object in visual space coordinates and units. These properties are *unencoded* in that they are intrinsic to the object and always present, but not normally available to the cognitive processor. That is, if there is a Location and Size property available to the cognitive processor, it would only be a result of some processor encoding the geometric values into a symbolic form, such as "Near" or "Large." Other properties such as Color and Shape are optional properties. A physical visual object resides in the Visual_physical_store and has a physical name assigned by the device, a geometric location and size, and some number of physical properties specified by the device, such as Color or Shape. The device can change the status or properties of an existing physical object by specifying its physical name.

A psychological visual object resides in each of the visual stores: the Visual_sensory_store and the Visual_perceptual_store. It also has a name, a geometric location and size, and some number of properties. The psychological name is assigned by the Eye_processor when the object first appears and is visible. The name has the form *Vpsychobj*n, where *n* is a unique number, ensuring that all psychological visual objects have a unique name. The properties of a visual object in Visual_sensory_store are determined by the Eye_processor, depending for example, on whether the shape of an object in peripheral vision can be seen. The properties of a visual object in Visual_perceptual_store are determined by the Visual_perceptual_processor, depending, for example, on how long the shape of an object formerly in foveal vision is still available once it is in peripheral vision.

Notice that once the appearance of an object has been processed through the visual system, there will be four representations of it: One in the physical store, another in sensory store, and third and a fourth in the perceptual store and the production system memory. The first three may have different properties (and thus must be represented as separate entities). The fourth is "slaved" to the third, and simply represents a change in representation to serve the purposes of the production system interpreter. Thus for theoretical purposes, the psychological visual objects are only those in sensory store and in perceptual store.

The profound problem of maintaining object identity is by-passed, or "finessed", in the current system. A psychologically complete system would explain how under certain conditions visual objects that temporarily disappear, or change location or other visual properties, are assumed by the human cognitive system to be the same object. To avoid getting bogged down in this very difficult problem, object identity is maintained by brute force with the assistance of the device, which knows the veridical state of the visual scene: The device assigns a physical name to the object and that name is specified in the input to the Visual_physical_store and subsequently to the Eye_processor. If the object is new, the Eye_processor assigns a psychological name to the object. If the device assigns a new property value to the object, such as changing its location or color, the device specifies this change using the physical name of the object. The Eye_processor looks up the psychological name corresponding to the physical name. All processors "downstream" of the Eye_processor thus use only the psychological name to refer to the object. As long as the device uses the same name to refer to an object, the visual system will treat it as the same object.

Retinal Availability

A fundamental feature of human vision is that the retina is not homogeneous (see Findlay & Gilchrist, 2003). The acuity of vision is highest in the fovea, and declines as the object moves out toward peripheral vision. The

conventional measurement of retinal position is the *eccentricity* - the distance on the retina (in degrees of visual angle) between the center of the fovea and the object in question. In addition to the eccentricity, several other factors govern whether a particular visual property can be detected: the size of the object, the density of surround objects (*lateral masking* or *flanker effects*) and the specific property (e.g. color vs. shape). The term *retinal availability* is used to refer to the detectability or recognizability of a particular visual property at a particular location on the retina.

The previous version of EPIC used a simple zone model of retinal availability. More elaborate models are currently under development.

Zone Availability

In this simple model of retinal availability, each visual property is said to be completely available within a certain eccentricity and completely unavailable outside that eccentricity. These zones can be specified separately for each visual property, but the default zones and their radii are as follows:

Fovea: 1 degree
Parafovea: 7.5 degrees
Periphery: 60 degrees

The assumption of fixed sizes for the zone is clearly approximate, especially in the case of the parafovea, whose size should be considered a free parameter, given the impoverished state of the empirical literature on exactly what perceptual properties are perceivable as a function of eccentricity.

The largest zone, the peripheral zone, which defines the outer limit of visibility of an object, is in fact "hard wired" into the Eye_processor.

Visual input from the simulated device

The device processor creates a visual object by calling the the make_object_appear function of the Human_processor. Human_processor acts as a wrapper around the entire psychological mechanisms of the simulated human - all external input arrives at Human_processor. Additional properties of the object can be specified or modified by calling other Human_processor functions like set_visual_object_property. By calling this functions, the Device causes visual input to be immediately supplied to the human - no time delays are involved between the Device and the Human_processor. The current list of these functions specify how visual objects can be created, deleted, and modified by the device processor:

void make_visual_object_appear(const Symbol& obj_name, GU::Point location, GU::Size size);

void make_visual_object_disappear(const Symbol& obj_name);

void set_visual_object_location(const Symbol& obj_name, GU::Point location);

void set_visual_object_size(const Symbol& obj_name, GU::Size size);

void set_visual_object_property(const Symbol& obj_name, const Symbol& propname, const Symbol& propvalue);

These functions are called by the device processor code. Each requires a name for the object, which is its physical name. Creating an object requires specifying its size and location, which are thus intrinsic and mandatory properties for a visual object. These two properties can be modified in other functions. Making an object disappear requires only its name. All other properties of the object, which would include common properties such as color or shape, are specified or changed with the set_visual_object_property function.

At this time, the Human_processor functions do nothing more than call the corresponding functions of the Visual_physical_store.

Visual_physical_store

The Visual_physical_store in essence represents the pattern of stimulation impinging on the simulated human's eyes. The purpose of having an independent representation of this information as the first stage in the human system is so that the Device processor will not be responsible for maintaining this information in a way that ensures that the simulated human works properly. Instead, it can generate the visual events in whatever manner is convenient - for example, the Device does not necessarily have to maintain any kind of data store of visual objects of its own, or if it does, it can maintain them in a manner convenient for the device simulation without regard to how they are specified to the Human_processor's components. Thus the interface between the Device and the Human_processor decouples

how the Device code is written from how the simulated human works.

Input arriving at the Visual_physical_store causes an immediate change in the contents of the physical store (such as changing the physical location of the object), and if the result is indeed a change in the state of the object, the input is simply forwarded to the Eye_processor in the form of a call to the corresponding function. No modification of the input is made, and no time delay is imposed on it.

The Eye_processor will access the contents of the Visual_physical_store as needed to obtain the physical properties of an object. For example, if the eye is moved to a new location where the shape of an object might now be available, the Eye_processor will access the object in the physical store to obtain its shape. The only constraint is that any changes to an object's location, size, and properties, or a call to make an object disappear, must refer to any existing object - one created as a result of an earlier call to make_visual_object_appear.

Eye_processor

Eye_processor represents the eye and the retina, and is one of the more complex processors in EPIC. The basic operation is to filter visual events by the retinal availability, and pass the resulting events to the Visual_sensory_store, in which the basic sensory properties of a visual object are represented. These events arrive at the sensory store after a transduction delay, which can be a different value for each visual property. The psychological name for a physical object is assigned by this processor, and used to refer to the visual object as it is processed through the visual system. Except for make_object_appear, it is a run-time error if any input refers to an unknown object.

An additional input type for the Eye_processor are eye movement events generated by the ocular motor processors. These events come in pairs: A *start* event marks the beginning of the movement, and an *end* event marks the end of the movement. The eye has a current location, which is a point in visual space representing the location of the center of the fovea. This location is changed to the new location when an end eye movement event arrives. Thus the eye always moves in jumps, but the size of the jump depends on the type and size of the eye movement: there are saccade eye movements, and smooth eye movements in which the jumps are considerably smaller. Either of these two types of eye movements can be produced by the involuntary ocular processor; the voluntary ocular processor produces only saccade eye movements.

Saccadic suppression is represented in a simple form: If an visual input event arrives while a saccadic eye movement is underway (the start event has arrived, but not the end event), the input event is rescheduled to arrive 1 ms after the eye movement is complete. Thus saccadic suppression results in a delay in recognition of a change, not in a loss of change information. This approach is necessary at this time because of the event-driven nature of the simulation system. That is, if a change event that arrives during an eye movement were simply discarded, there would be no natural way for the change in the visual object to then get noticed and passed into the visual system. Of course, the phenomena of change blindness suggest that exactly this might happen, so this approach might be modified in the future.

Finally, in addition to sending visual information into the rest of the system, the Eye_processor controls involuntary eye movements by commanding the involuntary oculomotor processor, Invol_ocular_processor, to make eye movements under certain conditions that can be enabled and disabled by cognitive production rule actions. If *Reflex* eye movements are enabled, then an onset or a movement of an object will trigger an eye movement to the object. If *Centering* is enabled, involuntary movements will be made to keep an object centered in the fovea. Details on both of these mechanisms appears below.

The following section describes what the Eye_processor does in response to each input function call or event that it receives. This description is the assumed psychological mechanism for this processor. The other processors will be described similarly.

This processor is somewhat unusual in that it receives inputs both in the form of function calls from the Visual_physical_store and in the form of events that are sent to it both by the Ocular motor processors and also by itself. These *send-to-self* events are how saccadic suppression is handled.

Input and Event handling:

make_object_appear

The new object carries a physical name, and a location and a size. It is added to the Visual_physical_store. A psychological name is created, and a record is kept that associates these two names with each other. Any future inputs with that same physical name are assumed to apply to the corresponding psychological name. If a saccade is underway, a Visual_Appear_event containing the appearance information is send back to the Eye_processor, scheduled to arrive at the eye movement end time + 1 ms, and nothing further is done. This

is an example of a send-to-self event.
The eccentricity of the new object is calculated; if it is not visible (its eccentricity is larger than the peripheral radius), nothing further is done. If it is visible, a Visual_Appear_event is scheduled to arrive at the Visual_sensory_store after Appearance_transduction_time has elapsed. This event, like all others sent to the Visual_sensory_store, uses the psychological name to refer to the object.
If reflexes are both enabled and *on*, and the eye is free to move, the Eye_processor sends an Invol_ocular_Saccade_action command to the involuntary ocular processor, scheduled to arrive after a delay of Appearance_transduction_time + Inform_invol_ocular_delay.

Visual_Appear_event
This sent-to-self event arrives at Eye_processor only as a result of a call to make_object_appear during an saccade, and it results in the same processing starting with calculating the eccentricity of the new object.

make_object_disappear
If the object is not visible, nothing further is done.
If the object was closest to the fovea center, and centering was active, centering is made inactive and the closest object information is discarded, and reflexes are turned on if they are enabled. This is to allow the eye to readily move to another object if the disappearing one was being centered.
After these checks, if a saccade is underway, a sent-to-self Visual_Disappear_event is scheduled to arrive at the Eye_processor and nothing further is done.
Otherwise, a Visual_Disappear_event is schedule to arrive at Visual_sensory_store after Disappearance_transduction_time.

Visual_Disappear_event
This sent-to-self event arrives only as a result of a call to make_object_disappear during an saccade. The processing done is the same scheduling of a Visual_Disappear_event to arrive at Visual_sensory_store after Disappearance_transduction_time.

set_object_location
If a saccade is underway, a sent-to-self Visual_Change_Location_event is scheduled and nothing further is done.
If the object is now not visible, nothing further is done.
Otherwise, a Visual_Change_Location_event is scheduled to arrive at Visual_sensory_store after Location_change_transduction_time, and the availability of all visual properties is updated.
If reflexes are both enabled and *on*, centering is not active, and the eye is free to move, the Eye_processor sends an Invol_ocular_Saccade_action command to the involuntary ocular processor, scheduled to arrive after a delay of Location_change_transduction_time + Inform_invol_ocular_delay.
The closest object to fovea center is updated because this object's moving might have changed the object that should be centered, and centering is performed.
The geometric location of an object in visual space is an intrinsic property - see above discussion of visual objects.

set_object_size
If a saccade is underway, a sent-to-self Visual_Change_Location_event or Visual_Change_Size_event is scheduled and nothing further is done. If the object is not visible, nothing further is done.
Otherwise, a Visual_Change_Size_event is scheduled to arrive at Visual_sensory_store after Size_change_transduction_time.
Because the size of an object can affect the availability of visual properties, the availability of all visual properties is updated.
The geometric size of an object in visual space is an intrinsic property - see above discussion of visual objects.

Visual_Change_Location_event, Visual_Change_Size_event
These sent-to-self events arrive as a result of these changing originally being made during a saccade. These unencoded properties are handled differently from the encoded properties. When an object changes location or size, this produces a short-lived clause in production system memory. The production system memory is immediately updated with a clause of the form:
(Visual <object name> Location Change) or (Visual <object name> Size Change)
After the parameter value Change_decay_time has elapsed, this clause is removed from the production system memory.

Possible change note: These events should probably be handled the same way as onset and offset events.

set_object_property

If a saccade is underway, a sent-to-self Visual_Change_Property_event is scheduled and nothing further is done.

Otherwise, the availability function for the property is consulted. It is an error if there is no availability function specified for the property. If the property is not available, a record is kept with the object representation in the Visual_physical_store, and nothing further is done.

If the property in an inter-object relation, such as Right_of, the value of the property is the physical name of the other object; the psychological name is looked up and substituted.

At this time, these inter-object relation properties are:

Left_of
Right_of
Above
Below
Placed_on

The time delay specified by the availability function for the property is determined, and a Visual_Change_Property_event is scheduled to arrive at the Visual_sensory_store after that time.

Visual_Change_Property_event

This sent-to-self event arrives only as a result of a call to set_object_property during an saccade; the processing is identical to that of set_object_property.

Eye_Voluntary_Saccade_Start_event, Eye_Voluntary_Saccade_End_event

These events are sent by the voluntary ocular processor. As described above, when the start event arrives, the only effect is to make saccadic suppression in effect. The eye does not actually change point of fixation until the end event arrives. At this point, if centering is enabled, and the type of eye movement calls for centering to be turned on at the end of the movement, centering is made active; otherwise off.

If reflexes are enabled, and the type of eye movement calls for turning on reflexes at the end of the movement, then reflexes are turn on; otherwise, off.

After completion of an eye movement, the state of all object locations and property availabilities needs to be updated. First the eccentricities of all objects are updated; then the availability of all properties, and finally the identity of the object closest to fovea center.

Centering is then performed in case the object has moved during the programming and execution of the eye movement.

Eye_Involuntary_Saccade_Start_event, Eye_Involuntary_Saccade_End_event

These events are sent by the involuntary ocular processor. The effect of an involuntary saccade is identical to that of a voluntary saccade except that after updating all of the objects, if centering is enabled, it is always made active, and centering is then performed.

Eye_Involuntary_Smooth_Move_Start_event, Eye_Involuntary_Smooth_Move_End_event

These events are sent by the involuntary ocular processor. Unlike saccades, there is no suppression of visual events during a smooth eye movement. All objects are updated after the end event, but no changes are made to the centering or reflex states, nor is centering performed separately - any motion of the closest-to-center object will produce subsequent eye movements.

Functions performed in response to inputs

Determining object visibility

At this time, the Eye_processor halts with an error if an object that was visible becomes invisible due to its eccentricity becoming too large - the code simply does not handle this case at this time. Note that this is different from an object disappearing while it is visible, which is handled as a visual disappearance event. In contrast, if the object is out of view, its status is unknown and it is not clear how it should be treated if it become visible again. Rather than speculate, at this time the code treats this situation as an error. On the other hand, if the object is visible, an Eccentricity property is synthesized and sent to the Visual_sensory_store as a Visual_Change_Property_event scheduled to arrive after a delay of Eccentricity_transduction_time. At this time, the Eccentricity property has two possible values: Fovea, if it is within the standard fovea radius of 1 degree, and Periphery otherwise.

Updating object properties

The availability of each property is computed using the availability function that is in effect for that

property. If there is no change in the availability of the property, then nothing further is done for that property. If the property has changed availability, then a Visual_Change_Property_event is scheduled to arrive at the visual sensory store after a delay specified by the availability function. If the property has become available, the change event contains the value of the property; if it is now unavailable, the change event contains the value *nil*, which represents no value; this causes the previous property/value pair to be removed from visual sensory store and visual perceptual store.

Determining visual availabilities

The availability specification for a property contains two functions: one returns true if the property of an object in the physical store is available retinally, false if not. The other returns the transduction time delay used when the object is available to determine when the event containing the property value should arrive at the visual sensory store.

These functions have access to all objects in the Visual_physical_store, and so can compute the availability (or transduction time) for an object as a function of the entire contents of visual scene if desired. The specific availability functions already programmed in the system can be supplied with parameter values to allow custom specification by the modeler; at this time, research is underway to determine a good "default" set of availability functions. These specifics are documented in the parameters section.

The availability functions take an additional parameter which is used to produce some variability in the results of applying the function. At this time, the availability function variability is expressed in terms of variability of the eccentricity of an object. Thus the value of Eccentricity_fluctuation_factor can be randomly sampled and supplied to the availability function which can use it to alter the availability value. For example, for simple zone availability, in which a property's availability is specified only in terms of eccentricity (e.g. that color is available within 5 degrees), the actual eccentricity is multiplied by the fluctuation factor to determine the effective eccentricity used to determine eccentricity. However, an availability function is not required to use this fluctuation, and can perform a different randomization instead. A similar fluctuation factor, Property_delay_fluctuation_factor, is supplied to the function that determines the transduction time delay.

Performing centering

Nothing is done if centering is either disabled or not active or the eye is not free to move.

If the closest object to fovea center is within the Foveal_bouquet_radius (the most central part of the fovea), nothing further is done - an object is already centered.

Otherwise, the Eye_processor sends a movement command to the involuntary ocular processor scheduled to arrive after a delay of Location_change_transduction_time + Inform_invol_ocular_delay and makes centering active. This command is either a Invol_ocular_Smooth_move_action if the closest object is within Centering_smooth_move_radius, or a Invol_ocular_Saccade_action if it is larger but within Centering_saccade_radius. If the closest object is too far away, centering is made inactive, and if reflexes are enabled, they are turned on. Thus if the eye "loses" what it is centering, it can make an eye movement to an object that either appears or moves.

Determining whether the eye is free to move

The eye is not free to move if any of the following are true:

- The last involuntary oculomotor command has not arrived at the motor processor;
- A voluntary saccade, involuntary saccade, or smooth movement, is underway;
- Either the voluntary or involuntary oculomotor processors are in the modality-busy state.

Current parameter values

Appearance_transduction_time: 20 ms
Eccentricity_transduction_time: 50 ms
Disappearance_transduction_time: 50 ms
Location_change_transduction_time: 20 ms
Size_change_transduction_time: 20 ms
Eccentricity_fluctuation_factor: mean 1.0, sd .1, Normal distribution
Property_delay_fluctuation_factor: mean 1.0, sd .5, Normal distribution
Foveal_bouquet_radius: .25 degree
Centering_smooth_move_radius: 1.0 degree
Centering_saccade_radius: 10.0 degree
Some basic "standard" values for availability functions
standard_fovea_radius: 1.0
standard_parafovea_radius: 7.5
standard_peripheral_radius: 60.0

standard_short_delay: 25
standard_delay: 50
standard_long_delay: 100

Properties recognized in availability functions - currently all are "Zone" availabilities

Relations (Position, Orientation, Above, Below, Right_of, Left_of, Placed_on, Depth,Distance)
Zone available within standard_peripheral_radius after standard_delay

Color
Zone available within standard_parafovea_radius after standard_delay

Shape
Zone available within standard_parafovea_radius after standard_long_delay

Text
Zone available within standard_fovea_radius after standard_long_delay

Size (encoded)
Zone available within standard_fovea_radius after standard_delay

Specifying availability parameters

This is a complex parameter whose syntax is specific to this processor. The syntax of an availability parameter specification is:

(Eye Availability <property> <availability_function_type> <function parameters> **)**

The <property> is a visual property name, such as Color, Shape, Text, etc. The first function parameter is normally a <delay> value; if the property is available, it has a transduction delay of <delay> * Property_delay_fluctuation_factor.

Availability Functions

The <availability_function_type> is the name of an availability function supplied in the Retina_functions module of the EPIC software. The alternative functions are still a matter for research, but some current representative ones and their parameter syntax are:

Zone <delay> <radius> - if the object center's eccentricity is within the supplied <radius>, the property is available; the specified radius must be between standard_fovea_radius and standard_peripheral_radius. This means that for Zone availability, all properties are available within standard_fovea_radius.

Exponential <delay> <coeff> <base> <exponent> - the property is available if the object's size is greater than the threshold size, which is an exponential function of the object's eccentricity, computed as:

threshold_size = <coeff> * <base> ^ (<exponent> * eccentricity)

where:

eccentricity = actual eccentricity * Eccentricity_fluctuation_factor

The object's size is the average of its horizontal and vertical sizes.

Flat <delay> <probability> - the property is available with the specified probability, regardless of the object's size or eccentricity, but it is always available if the object is within the standard_fovea_radius. This is primarily useful for data fitting exploration.

Visual_sensory_store

The visual sensory store is fairly close to "iconic memory" as it has been presented in the psychological literature. Basically it acts as a buffer between the Eye_processor and the perceptual processor. As input events arrive from the Eye_processor, the contents of the store are updated, and the events then immediately passed on the Visual_perceptual_processor in the form of calls to its functions. At this time, Visual_sensory_store is a very simple system: it is slaved to the current output of the eye, but because information is retained for some time after it disappears from the visual field or becomes unavailable from the retina, it lags behind.

The visual objects in this store represent the current contents of the visual field as filtered by retinal availability. At this time, there is no limit to the amount of information that can held in the Visual_sensory_store.

Event handling:

Visual_Appear_event

The new object with the supplied name, location, and size is added to the store. The Visual_perceptual_processor is immediately notified of the change with a call to its make_object_appear function.

Visual_Disappear_event
A note is made for the benefit of concerned parts of the software (e.g. the Epic_View for the sensory store) that the object is disappearing, but no actual change is made to its state yet. A Visual_Erase_event is scheduled to arrive after Disappearance_decay_delay_time.

Visual_Erase_event
The Visual_perceptual_processor is notified that the object is gone by a call to make_object_disappear, and the object is removed from the sensory store.

Visual_Change_Location_event, Visual_Change_Size_event, Visual_Change_Property_event
The object in the store is updated using the supplied information. If the value of the property is actually different, then the Visual_perceptual_processor is immediately informed of the change.

Current parameter values
Disappearance_decay_delay_time: 200 ms

Visual_perceptual_processor

This processor represents the processes that encode the sensory information about objects in the Visual_sensory_store and pass on the encodings to the Visual_perceptual_store. At this time, the processor does nothing more than impose a time delay for the encoding of specific visual events and properties. Unfortunately, the current version does not allow a modeler to easily supply an arbitrary perceptual encoding function that will respond to any desired pattern of visual properties with an encoding, as in the previous LISP version of EPIC. The capability for this is present in the current software, and it can be done by anyone willing to modify the code, but a fully supported means is not yet available.

When events arrive at the Visual_sensory_store, they are passed on immediately to the Visual_perceptual_processor, which then creates events that arrive at the Visual_perceptual_store at a future time. Thus the basic action this processor is simply to create new events in response to changes in the contents of the Visual_sensory_store.

Input handling:

make_object_appear
A Visual_Appear_event is scheduled to arrive at Visual_perceptual_store after parameter value Appearance_recoding_time. A detection onset event is created and transmitted as a Visual_Change_Property_event with property name Detection and value Onset to arrive at the same time, and then this property will be deleted at a later time with a Visual_Change_Property_event with property name Detection and value nil to arrive after parameter value Onset_decay_time.

make_object_disappear
A Visual_Disappear_event is scheduled to arrive at Visual_perceptual_store after parameter value Disappearance_recoding_time. A detection offset event is created and transmitted as a Visual_Change_Property_event with property name Detection and value Offset to arrive at the same time, and then this property will be deleted at a later time with a Visual_Change_Property_event with property name Detection and value nil to arrive after parameter value Offset_decay_time.

set_object_property
This input is supplied when an encoded property such as Color changes value. A Visual_Change_Property_event is scheduled to arrive at Visual_perceptual_store after a recoding time. The recoding time for each property (e.g. Color) is looked up in a table and multiplied by the parameter Recoding_time_fluctuation_factor which can be set to fluctuate randomly to give some variability to the recoding times. The individual property recoding times can be set in a production rule file using the parameter name Recoding_time followed by the property name and the recoding time delay.

set_object_location, set_object_size
These two non-encoded properties result in the creation of the corresponding Visual_Change event being sent to the Visual_perceptual_store after a delay of Location_change_recoding_time or Size_change_recoding_time.

Current parameter values
Appearance_recoding_time: 25 ms
Disappearance_recoding_time: 50 ms
Onset_decay_time: 75 ms
Offset_decay_time: 75 ms

Location_change_recoding_time: 25 ms
Size_change_recoding_time: 25 ms
Recoding_time_fluctuation_factor: 1.0, Fixed
Recoding_time for individual properties:
Eccentricity, Color, Encoded Size, Shape: 50 ms
Position, Left_of, Right_of, Above, Below, Distance: 50 ms
Text, Orientation, Depth: 100 ms

Specifying recoding time parameters

This is a complex parameter whose syntax is specific to this processor. The syntax of a recoding time parameter specification is:

(Visual_perceptual_processor Recoding_time <property> <time> **)**

The <property> is a visual property name, such as Color, Shape, Text, etc. The <time> value is used to compute the recoding time as:

<time> * Recoding_time_fluctuation_factor

Visual_perceptual_store

The visual perceptual store plays the role often called "visual working memory," but as has often happened in psychological theory, the structure or mechanism is confused with the experimental paradigms used to assess it; this component of EPIC thus does not conform to any single one of the notions that have been termed visual working memory.

The visual objects in this store represent the current contents of the visual field as they are available to the cognitive processor. Thus the Visual partition of the production system memory is slaved to the contents of the visual perceptual store. When an object appears, it and its properties soon become available depending on the transduction and encoding delays specified in the Eye processor and visual perceptual processor. However, when the input information become unavailable due to eye movements, or an object disappears, it persists in both the sensory store and the perceptual store for some time. If an object disappears from the perceptual store, all of its properties also disappear. However, it is possible for a property to be lost while the object and some of its other properties are still available - for example, as a result of changes in eccentricity due to object motion or eye movements.

A visual object in perceptual store can be in one of three states: Present, Unsupported, or Retained. Present means that the object is currently known to be present in the visual scene. Unsupported means that the object is known to have disappeared from the visual scene, so its current representation is no longer supported by actual visual input, but is persisting for some time in its last state. Retained means that the object is now strictly in memory, where it will be retained for some time. Distinguishing between Unsupported and Retained states makes it possible in the future to develop models for visual working memory in which all objects persist for some time in the Unsupported state, but only selected objects persist for a longer time in the Retained state. Similarly, but simpler, each of its encoded properties (e.g. Color, Shape) can be in a present or unsupported state. Each event refers to an object by its psychological name, and some events include the name and value of an encoded visual property.

This store is locus of any hypothesized time and capacity limitations on how many objects or how much information about object can be retained and made available to the Cognitive_processor, or how objects in the store might interfere with each other. Such limitations would be implemented in the code that handles events for object and property appearance and disappearance. However, at this time, the objects in visual perceptual memory are independent of each other and there there is no overall capacity limit to the Visual perceptual store. The current rationale for this is that since this store represents the current visual scene, it should be able to represent a rather large number of objects and amount of information. The results that suggest a rather small capacity of visual working memory all deal with situations in which the objects in question are no longer present, and often when other similar objects are present. This suggests a limitation in how many objects can be in the Unsupported or Retained state, rather than an overall limitation on the capacity of the perceptual store for Present objects or properties. Pending resolution of this problem, no overall limit is set on the perceptual store, although we have experimented with a limit on the number of Unsupported or Retained items; this experimental code has been removed in the current version.

Event handling:

Visual_Appear_event

The new object with the supplied name, location, and size is added to the store. The Cognitive_processor production system memory is immediately updated with a clause of the form:

(Visual <object name> Status Visible)

to notify the cognitive processor that an object is now present and visible. This Status clause will change when the object disappears. Note that this Status is different from the Detection Onset event that arrives separately.

Visual_Disappear_event

When an object disappears, the information is lost according to a series of subsequent events; the object is first put in the Unsupported state - it and its properties are no longer supported by sensory input. A Visual_Erase_event is scheduled to arrive after parameter value Unsupported_object_decay_time has elapsed. Note that no change is made to the object information in production system memory, so during the Unsupported time, the object is fully present as far as the Cognitive_processor is concerned.

Visual_Erase_event

If the object was in the Unsupported state, it now goes into the Retained state. A new Visual_Erase_event is scheduled to arrive after parameter value Retained_object_decay_time. The Cognitive processor is informed that the object is disappearing with the Status of the object changing to:

(Visual <object name> Status Disappearing)

If the object was in the Retained state, it is now time for it to be completely removed from the perceptual store and the production system memory. All clauses in the Visual production system memory that refer to the object are immediately removed, which includes the Status clause and each clause referring to its encoded properties. Since the psychological object is now completely gone, the table maintained by the Eye_processor that relates physical names to psychological names is updated to remove both names; this ensure that if the device creates a new object with the same physical name, it will be treated as a new object by the Eye_processor.

Visual_Change_Property_event

When an encoded property changes, it could be due to a change originating in the physical object (e.g. it changes color) or to a change in eccentricity that renders that property available or unavailable. If the property becomes unavailable, it becomes Unsupported and then erased. Since a property might become unavailable, and then available, arrangements are made to ensure that a new available value overrides a disappearance of the property. More specifically:

If the visual property is now unavailable (indicated by a new value of nil), a Visual_Erase_Property_event is scheduled to arrive after parameter value Property_decay_time and a record is made that the property is Unsupported and expected to be erased at that time, and nothing further is done.

If the property in question is Detection, it is ignored - the appearance and disappearance of this property is handled separately.

If the visual property is being changed to a new available value, any previous record of Unsupported and expected erasure is deleted. Then the perceptual store and the Cognitive_processor production system memory are immediately updated to remove the property or change its value.

Visual_Erase_Property_event

If the event applies to a property whose erasure was canceled, or specifies the erasure of a different value, this event is ignored - the property of the object became supported again due to its becoming available again, perhaps with a different value. Otherwise, the property is removed from the object representation in the perceptual store, and the Cognitive_processor immediately removes the production system memory clause of the form:

(Visual <object name> <property name> <last property value>)

Visual_Change_Location_event, Visual_Change_Size_event

These unencoded properties are handled differently from the encoded properties. When an object changes location or size, this produces a short-lived clause in production system memory. The production system memory is immediately updated with a clause of the form:

(Visual <object name> Location Change) or (Visual <object name> Size Change)

After the parameter value Change_decay_time has elapsed, the this clause will be removed from the production system memory.

Possible change note: These events should probably be handled the same way as onset and offset events.

Current parameter values

Change_decay_time: 75 ms

Unsupported_object_decay_time: 200 ms

Retained_object_decay_time: 500 ms
Property_decay_time: 500 ms

The Auditory System

Auditory Representation and Processing

Auditory Processing Overview

As shown in Figure 3, the auditory system roughly parallels the visual system in having a somewhat traditional series of stores and processors. However, the theoretical development of the auditory system in the literature lags that of the visual system, and thus EPIC's representation of auditory perception has a certain amount of theoretical novelty - while the visual system deals only with one kind of object, the auditory system has two - one for sounds, and the other for the sources of these sounds. Sources have physical locations, but no auditory properties of their own, while sounds are always associated with a source and have auditory properties. Thus a source can be responsible for a sequence of sounds, or even simultaneous but different sounds.

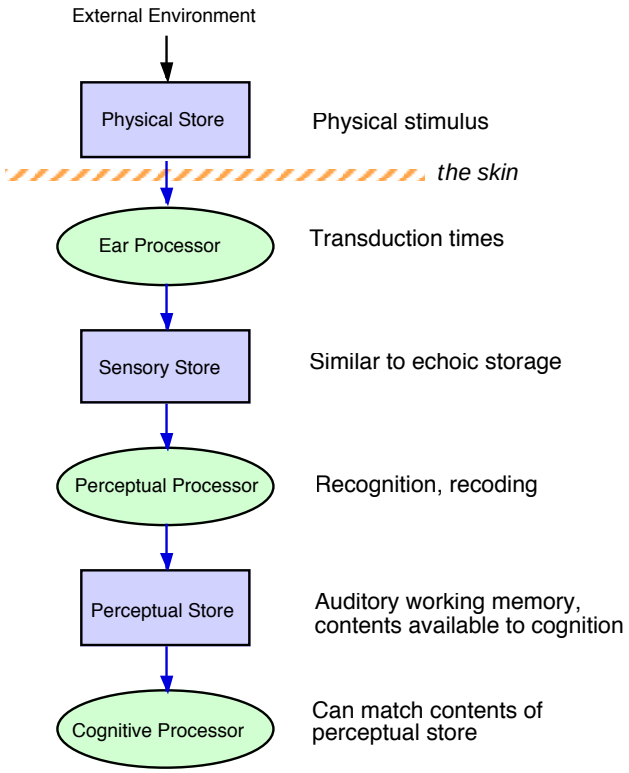

Figure 3. EPIC's auditory system - compare with the visual system.

The stores and processors deal with two kinds of auditory objects: An *auditory source* is a source of *auditory sounds*. An auditory source is assumed to be an physical object located in space, and so has a location and size (similar to visual objects) as well as a name and other properties. It also has a creation time. The location and size are specified in terms of visual angle units. A default size of a point source is provided. There is a default source, located at (0,0), corresponding to using monophonic headphones.

An auditory sound is required to come from an auditory source, and has a mandatory property of timbre, which is the basic sound characteristic (e.g. a buzz versus a beep). In addition to other properties such as pitch, a sound has a duration and an intrinsic duration. If a duration is unspecified, the sound is taken to be continuing. The intrinsic duration is the minimum duration of the sound logically required to categorize it. For example, a sustained oboe note will be recognizable as having a certain pitch and timbre and thus being an oboe after a fairly short time, but might continue for a long time. A sound also has a creation time. Sounds are sequential in time, so each sound carries a symbolic link to the next sound. A sequence of sounds from the same source corresponds to a stream in Bregman's (1990) sense, however, at this time, there is no provision for a source being synthesized from sound sequences independently of their physical source, which is a great concern in Bregman's work.

At this time, there are possibly egregious differences in when certain similar operations are performed in the auditory and visual systems. In addition, at this time, the conditions under which sources should be created and

destroyed, and what additional properties they might have is unclear - the architecture is currently incomplete and underspecified in this regard.

Symbolic representation

As with vision, the EPIC architecture includes no mechanisms for performing the actual pattern recognition involved in audition or speech processing. Thus auditory inputs are symbolically coded collections of property-value pairs that specify the sensory and perceptual attributes of the stimulus.

Auditory input from the simulated device

The device processor creates a auditory source by calling the the create_auditory_source function of the Human_processor, and creates a sound by calling a set of functions such as make_auditory_sound_start. Human_processor acts as a wrapper around the entire psychological mechanisms of the simulated human - all external input arrives at Human_processor. Additional properties of the object can be specified or modified by calling other Human_processor functions like set_auditory_source_location. By calling this functions, the Device causes auditory input to be immediately supplied to the human - no time delays are involved between the Device and the Human_processor. The current list of these functions specify how auditory sources and sounds can be created, deleted, and modified:

```
void create_auditory_source(const Symbol& name, GU::Point location, GU::Size size = GU::Size());

void destroy_auditory_source(const Symbol& name);

void set_auditory_source_location(const Symbol& name, GU::Point location);

void set_auditory_source_size(const Symbol& name, GU::Size size);

void set_auditory_source_property(const Symbol& name, const Symbol& propname, const Symbol&
propvalue);


void make_auditory_sound_start(const Symbol& name, const Symbol& timbre,
      const Symbol& source, long intrinsic_duration);

void make_auditory_sound_stop(const Symbol& name);

void make_auditory_sound_event(const Symbol& name, const Symbol& timbre,
      const Symbol& source, long duration, long intrinsic_duration = 0);

void make_auditory_speech_sound_event(const Symbol& name, const Symbol& source,
      const Symbol& utterance, long duration);

void set_auditory_sound_property(const Symbol& name, const Symbol& propname, const Symbol& propvalue);
```

Each function requires a name for the object, which is its physical name. Creating an object requires specifying its size and location, which are thus intrinsic and mandatory properties for a visual object. These two properties can be modified in other functions. Making an object disappear requires only its name. All other properties of the object, which would include common properties such as color or shape, are specified or changed with the set_visual_object_property function.

The two "event" functions, make_auditory_sound_event and make_auditory_speech_sound_event are present only to simplify the programming of external device - these functions result in calls to the more basic functions that cause a sound to be started and stopped.

At this time, the Human_processor functions do nothing more than call the corresponding functions of the Auditory_physical_store.

Auditory_physical_store

The Auditory_physical_store represents the current physical state of the sound environment. As in the case of the visual system, the purpose of having an independent representation of this information as the first stage in the human system is so that the Device processor will not be responsible for maintaining this information in a way that ensures that the simulated human works properly. This store is responsible for maintaining the correspondence between physical and psychological names for sources and sounds

If a new source or sound is created, it is assigned a psychological name, and future references to the same physical

name will be mapped to the psychological name. Generally, input arriving at the physical store is simply forwarded to the Ear_processor; the specific details are presented next.

Input and Event handling:

create_source

The new source object carries a physical name, and a location and a size. It is added to the Auditory_physical_store. A psychological name is created, and a record is kept that associates these two names with each other. The psychological name has the form "Src*n*" such as "Src23". Any future inputs with that same physical name are assumed to apply to the corresponding psychological name, which is always sent forward. The create_source function of the Ear_processor is then called.

destroy_source

The source object is removed from the physical store, and the Ear_processor is informed, and the physical and psychological names are discarded.

set_source_location, set_source_size, set_source_property

After updating the source in the physical store, the corresponding call is made to the Ear_processor.

make_sound_start

For convenience in device programming, if the source is not identified (an empty string), the name DefaultSource is used, which is assigned the psychological name SrcDefaultSource. The source object is then created if not already present.

Once the source psychological name is available, a new sound object is created and added to the physical store, with a duration of zero to represent a continuing sound. The psychological name of the new sound object is recorded; it has the form "Snd*n*", as in "Snd42". The name of the future next sound object is determined; this is sent to the Ear processor along with the other parameters of the sound as the "next" link value to represent the serial order of the sounds.

make_sound_stop

The Ear_processor is informed that the sound is stopping, and the sound object is removed from the physical store, and the physical and psychological names are discarded.

set_sound_property

The sound object is updated, and the Ear_processor is called.

make_sound_event

make_sound_start is called, then an Auditory_Sound_Stop_event is scheduled to arrive at the physical store after the specified duration.

Auditory_Sound_Stop_event

The arrival of this event results in a simple call to make_auditory_sound_stop.

make_speech_sound_event

A new sound is added to the physical store, with a timbre of Speech, and a Content property consisting of the supplied utterance Symbol. A psychological name of the form "Aspeech" is assigned. The Ear_processor is notified of a new sound starting with timbre of Speech and Content of the utterance. An Auditory_Sound_Stop_event is then scheduled to arrive at the physical store after the specified duration.

Ear_processor

Unlike the Eye_processor, the Ear_processor is rather simple at this time, and simply passes events through to the Auditory_sensory_store after a transduction delay.

Input and Event handling:

create_source

An Auditory_Source_Create_event is scheduled to arrive at the auditory sensory store after Appearance_transduction_time.

destroy_source

An Auditory_Source_Destroy_event is scheduled to arrive at the auditory sensory store after Disappearance_transduction_time.

set_source_location, set_source_size, set_source_property

An Auditory_Source_Set_Location_event, Auditory_Source_Set_Size_event, or Auditory_Source_Set_Property_event is scheduled to arrive at the auditory sensory store after Location_change_transduction_time for location and size, and Property_transduction_time for all other properties.

make_sound_start
An Auditory_Sound_Start_event is scheduled to arrive at the auditory sensory store after Appearance_transduction_time. In the future, the involuntary ocular processor will be signaled to trigger a reflexive eye movement to the sound's source location, as in the final version of the LISP EPIC.

make_sound_stop
An Auditory_Sound_Stop_event is scheduled to arrive at the auditory sensory store after Disappearance_transduction_time.

set_sound_property
An Auditory_Sound_Set_Property_event is scheduled to arrive at the auditory sensory store after Property_transduction_time.

Current parameter values
Appearance_transduction_time: 25 ms
Disappearance_transduction_time: 100 ms
Location_change_transduction_time: 50 ms
Property_transduction_time: 100 ms

Auditory_sensory_store

The auditory sensory store is fairly close to "echoic memory" as it has been presented in the psychological literature. Basically it acts as a buffer between the Ear_processor and the perceptual processor. As input events arrive from the Ear_processor, the contents of the store are updated, and the events then immediately passed on the Auditory_perceptual_processor in the form of calls to its functions. If a source or sound disappears, it is retained for some time in the auditory store. At this time, Auditory_sensory_store is a very simple system; its characteristics need to be fleshed out considerably; for example, there is no representation of phonetic similarity effects on retention.

Event handling:

Auditory_Source_Create_event
The new object with the supplied name, location, and size is added to the store. The Auditory_perceptual_processor is immediately notified of the change with a call to its make_object_appear function.

Auditory_Source_Destroy_event
A note is made for the benefit of concerned parts of the software (e.g. the Epic_View for the sensory store) that the object is disappearing, but no actual change is made to its state yet. An Auditory_Source_Erase_event scheduled to arrive at the sensory store after Disappearance_decay_delay_time.

Auditory_Source_Erase_event
The perceptual processor is notified that the source object should be erased, and then it is removed from the sensory store.

Auditory_Source_Set_Location_event, Auditory_Source_Set_Size_event, Auditory_Source_Set_Property_event
The object in the store is updated using the supplied information. If the value of the property is actually different, then the Auditory_perceptual_processor is immediately informed of the change.

Auditory_Sound_Start_event
The sound object is created, and the auditory perceptual processor is notified.

Auditory_Sound_Stop_event
A note is made for the benefit of concerned parts of the software (e.g. the Epic_View for the sensory store) that the object is disappearing, but no actual change is made to its state yet. An Auditory_Sound_Erase_event scheduled to arrive at the sensory store after Disappearance_decay_delay_time.

Auditory_Sound_Erase_event
The perceptual processor is notified that the sound object should be erased, and then it is removed from the sensory store.

Auditory_Sound_Set_Property_event
The sound object is updated, and if the property value is different, the perceptual processor is notified.

Current parameter values

Disappearance_decay_delay_time: 200 ms

Auditory_perceptual_processor

In keeping with the incomplete state of the auditory system, the auditory perceptual processor is also rather simple at this time, and mostly just passes events through to the Auditory_perceptual_store after a recoding delay.

Input handling:

create_source

An Auditory_Source_Create_event is scheduled to arrive at the auditory perceptual store after Appearance_recoding_time.

destroy_source

An Auditory_Source_Destroy_event is scheduled to arrive at the auditory perceptual store after Disappearance_recoding_time.

set_source_location, set_source_size

An Auditory_Source_Set_Location_event, Auditory_Source_Set_Size_event, or Auditory_Source_Set_Property_event is scheduled to arrive at the auditory perceptual store after Location_change_recoding_time.

set_source_property

The recoding time for the specified property is looked up, multiplied by Recoding_time_fluctuation_factor, and an Auditory_Source_Set_Property_event is scheduled to arrive at the auditory perceptual store after that time.

make_sound_start

An Auditory_Sound_Start_event is scheduled to arrive at the auditory perceptual store after Appearance_recoding_time. In addition, an Auditory_Sound_Set_Property_event is scheduled to arrive at the same time that sets the property Detection to Onset; a second such event is scheduled to arrive after Transient_decay_time to remove the Detection property.

make_sound_stop

An Auditory_Sound_Stop_event is scheduled to arrive at the auditory perceptual store after Disappearance_recoding_time. In addition, an Auditory_Sound_Set_Property_event is scheduled to arrive at the same time that sets the property Detection to Offset; a second such event is scheduled to arrive after Transient_decay_time to remove the Detection property.

set_sound_property

The recoding time for the specified property is looked up, multiplied by Recoding_time_fluctuation_factor, and an Auditory_Sound_Set_Property_event is scheduled to arrive at the auditory perceptual store after that time.

Current parameter values

Appearance_recoding_time: 50 ms
Disappearance_recoding_time: 50 ms
Location_change_recoding_time: 50 ms
Transient_decay_time: 75 ms
Recoding_time_fluctuation_factor: mean 1.0, sd .5, Normal distribution
Property recoding times:
Pitch: 100 ms
Loudness: 100 ms
Timbre: 100 ms

Specifying recoding time parameters

This is a complex parameter whose syntax is specific to this processor. The syntax of a recoding time parameter specification is:

(Auditory_perceptual_processor Recoding_time <property> <time> **)**

The <property> is a auditory property name, such as Pitch, Timbre, etc. The <time> value is used to compute the recoding time as:

<time> * Recoding_time_fluctuation_factor

Auditory_perceptual_store

The auditory perceptual store plays a role very similar to the the visual perceptual store, in that it contains the sources and sounds currently in the auditory scene. However, few assertions can be made about its properties at this time, and it remains to be fleshed out.

The auditory objects in this store represent the current contents of the auditory scene as they are available to the cognitive processor. Thus the Auditory partition of the production system memory is slaved to the contents of the auditory perceptual store. When, e.g., a sound starts, it and its properties soon become available depending on the transduction and encoding delays specified in the Ear processor and auditory perceptual processor. However, when the sound stops, it persists in both the sensory store and the perceptual store for some time. If an object disappears from the perceptual store, all of its properties also disappear.

A visual object in perceptual store can be in one of three states: Present, Unsupported, or Retained. Similarly, but simpler, each of its encoded properties (e.g. Timbre, Pitch) can be in a present or unsupported state. Each event refers to an object by its psychological name, and some events include the name and value of an encoded visual property.

This store is locus of any hypothesized time and capacity limitations on how many sounds or how much information about a sound can be retained and made available to the Cognitive_processor, or how sounds in the store might interfere with each other. Such limitations would be implemented in the code that handles events for object and property appearance and disappearance. However, at this time, the objects in auditory perceptual memory are independent of each other and there there is no overall capacity limit to the auditory perceptual store; to conform to various results on verbal working memory, this needs to be corrected.

Event handling:

Auditory_Source_Create_event

The new source object with the supplied name, location, and size is added to the store. The Cognitive_processor production system memory is immediately updated with a clause of the form:

(Auditory <source name> Status Present)

to notify the cognitive processor that an object is now present - note that if there is currently no sound, the source won't be audible. This Status clause will change when the source disappears.

Auditory_Source_Destroy_event

At this time, when a source disappears, it simply persist for a time and is then removed. Thus, an Auditory_Source_Erase_event is scheduled to arrive at the perceptual store after Unsupported_decay_delay_time..

Auditory_Source_Erase_event

The source object is removed from the perceptual store, and the cognitive processor production system memory is updated to remove all clauses that were stored about the source object.

Auditory_Source_Set_Location_event, Auditory_Source_Set_Size_event

The object in the store is updated using the supplied information. If the value of the property is actually different, then the cognitive processor working is immediately informed of the change.

These properties are handled differently from the other properties of a source. When a source changes location or size, this produces a short-lived clause in production system memory. The production system memory is immediately updated with a clause of the form:

(Auditory <source name> Location Change) or (Auditory <source name> Size Change)

After the parameter value Change_decay_time has elapsed, the this clause will be removed from the production system memory.

Possible change note: These events should probably be handled the same way as onset and offset events.

Auditory_Source_Set_Property_event

The property value is immediately updated in the cognitive processor production system memory.

Auditory_Sound_Start_event

The new sound object with the supplied name and properties is added to the store. The Cognitive_processor production system memory is immediately updated with clauses of the form:

(Auditory <sound name> Status Audible)

(Auditory <sound name> Source <source name>)

(Auditory <sound name> Timbre <timbre>)

(Auditory <sound name> Next <next sound name>)

to notify the cognitive processor that a sound is now present and what its mandatory properties are. This Status clause will change when the sound disappears. Note that this Status is different from the Detection Onset and Offset events that arrives separately.

Auditory_Sound_Stop_event

When an sound disappears, the information is lost according to a series of subsequent events; the sound object is first put in the Unsupported state - it and its properties are no longer supported by sensory input. A Auditory_Sound_Erase_event is scheduled to arrive after parameter value Unsupported_decay_delay_time has elapsed. Note that no change is made to the object information in production system memory, so during the Unsupported time, the object is fully present as far as the Cognitive_processor is concerned.

Auditory_Sound_Erase_event

If the object was in the Unsupported state, it now goes into the Retained state. A new Auditory_Sound_Erase_event is scheduled to arrive after parameter value Retained_decay_delay_time. The Cognitive processor is informed that the sound is disappearing with the Status of the object changing to:

(Auditory <sound name> Status Fading)

If the sound object was in the Retained state, it is now time for it to be completely removed from the perceptual store and the production system memory. All clauses in the Auditory production system memory that refer to the sound object are immediately removed, which includes the Status, Source, Timbre, and Next clauses.

Auditory_Sound_Set_Property_event

If the property has been removed, it becomes Unsupported and then erased. Since a property might be removed temporarily, arrangements are made to ensure that a new available value overrides a disappearance of the property. More specifically:

If the sound property is now unavailable (indicated by a new value of nil), a Auditory_Sound_Erase_Property_event is scheduled to arrive after parameter value Property_decay_time and a record is made that the property is Unsupported and expected to be erased at that time, and nothing further is done.

If the property in question is Detection, it is ignored - the appearance and disappearance of this property is handled separately.

If the auditory sound property is being changed to a new value, any previous record of Unsupported and expected erasure is deleted. Then the perceptual store and the Cognitive_processor production system memory are immediately updated to remove the property or change its value.

Auditory_Sound_Erase_Property_event

If the event applies to a property whose erasure was canceled, or specifies the erasure of a different value, this event is ignored - the property of the object became supported again. Otherwise, the property is removed from the object representation in the perceptual store, and the Cognitive_processor immediately removes the production system memory clause of the form:

(Auditory <sound name> <property name> <last property value>)

Current parameter values

Change_decay_time: 75 ms
Unsupported_decay_delay_time: 200 ms
Retained_decay_delay_time: 1800 ms
Property_decay_time: 500 ms

The Tactile Processor

At this time, the C++ version of EPIC contains no tactile processor. In the LISP version of EPIC, it is very simple: It recodes its input as a TACTILE type output, and sends it to the cognitive processor with a time delay equal to the sum of its DETECTION-TIME and RECODING-TIME.

Motor Processing

Motor processor overview

Motor processors get inputs from the cognitive processor, resulting from the Send_to_motor action in production rules, carry out the commanded movements, and keep the cognitive processor updated about their state in the form of Motor clauses in the production system memory.

- The Manual_processor updates hand positions as necessary and sends various events to the Device_processor (e.g. which key was pressed at what time).
- The Ocular_processor and Invol_ocular_processor send eye movement events to Eye_processor.
- Vocal_processor sends speech events to the Device_processor.

In the current version of EPIC, the motor processor implementations share their basic structure and much of their code via inheritance from the class Motor_processor, and all of the movement-specific mechanisms are packaged in other classes inheriting from the class Motor_action. Thus the approach of this section is to present the basic behaviors and mechanisms that apply to all of the motor processors and movement types, and then describe the particulars for each movement modality and movement type.

The kinds of movement produced by a motor processor are described by a sets of features, the most important of which is the movement *style*, the basic kind of movement, such as Punch (a button) versus Point (at an object). The features for a Style appear in a hierarchy that governs which features must be generated when a given feature changes from a previous one. For example, changing the movement Style feature always requires that all subordinate features, such as Hand and Finger for Punch, be generated. However, some features are equal in subordination to another, meaning that changing one does not require regenerating the other.

Motor processors work in three phases, which can be overlapped to permit a series of movements to be made faster than if each movement must wait for the previous to finish before its processing can begin. A motor processor first generates a list of features for the movement, in the *preparation* phase. Each feature takes a certain amount of time to generate, currently assumed to be 50 ms. The movement *initiation* phase then requires additional time to set up the actual movement, also currently assumed to be 50 ms. The *execution* phase consists of making the actual mechanical motion; this takes an amount of time that depends on the movement.

There are two basic forms of the Send_to_motor action: *Perform* causes the commanded movement to be executed as soon as it is prepared and can be executed. *Prepare* causes the movement to be prepared, but execution is deferred. The next movement command can benefit from the previous feature preparation, and thus be ready to execute more rapidly. At this time, the current version of EPIC requires that a movement be fully specified for both Perform and Prepare commands - there is is no provision for partial specification of movements.

The motor processors have the ability to overlap the initiation and execution phase of a movement with the preparation phase for the next movement. The motor processor informs Cognitive_processor of its state. The steps involved are as follows:

1. The processor receives a movement command from Cognitive_processor to make a movement, and the preparation phase begins.
   - If another movement command arrives while preparation is in progress, the motor processor is "jammed", producing a run-time error and halting the simulation.
   - The new features are compared with the features from the last movement to see which features can be reused. Only features that are different contribute to the preparation time.
   - The preparation time is the parameter Feature_time * the number of features (including the Style). The Feature_time parameter is currently the same for all motor processors and movement styles.
   - When the preparation phase is complete, the movement then becomes a *ready* movement that can be executed.
   - If the movement command was *Prepare*, then no further processing is done with this movement - its execution is "deferred" - a later Perform command of the same movement will result in zero preparation time, or less preparation time for a similar movement.
2. If there is no movement in the execution phase, the ready movement enters the execution phase. First there is a movement initiation phase, which takes Initiation_time, a parameter currently shared by all motor processors and movement styles. After initiation, the physical execution of the movement is performed; this normally results in events being sent to other components of the simulation. The time required for the actual execution depends on

the type of the movement and its features.

- If a new movement gets prepared before the ready movement is executed, then the new movement replaces the previous ready movement. This permits the Cognitive_processor to quickly update a intended movements, but the previously commanded movement will be lost and not executed.
- Once the initiation and execution of a movement is begun, it runs through to completion; it is not possible to halt or interrupt it.
- Some movements involve actions at the end of the movement, such as updating hand positions.

3. When the movement execution is completed, and there is another ready movement waiting, it becomes the executing movement. If not, the the motor processor is finished with all commanded movements.

In order to allow production rules to control the motor processors properly, state indicator items are maintained in the production rule memory that reflect the current state of each motor processor. These items are of the form (Motor <modality> <phase> <state>), for example, (Motor Manual Modality Busy), where:

- <modality> is Ocular, Manual, or Vocal
  Note that Invol_ocular_processor, the Involuntary Ocular processor, does not produce signals - it is not subject to cognitive monitoring or direct control.
- <phase> is Modality, Processor, Preparation, or Execution, and corresponds to phases of movement execution
- <state> is Busy or Free

These phases and states are defined as follows:

- Modality becomes Busy when the command has been received by the motor processor, and then becomes Free when the movement execution is deemed complete. It indicates that the entire movement modality is working on a movement.
- Processor becomes Busy when the command is received, and then becomes Free when execution begins. It indicates the time that the motor processor is involved in preparing and "handing off" a movement for execution.
- Preparation becomes Busy as soon as the command is received, and becomes Free when the preparation is complete. It indicates when the feature generation mechanism is tied up; a new movement can be prepared as soon as it finishes, but the new movement might overwrite the previously commanded movement.
- Finally, Execution becomes Busy when the execution begins to be initiated, and Free when it is deemed complete. It indicates when the physical movement is starting or is underway.

The concept underlying this set of state signals can be described in terms of different cognitive strategies for when to send the next movement command to the motor processor: A production rule can wait for Preparation Free if it wants a movement executed as soon as possible, at the possible expense of overwriting an already prepared movement. If it is necessary to ensure that the previously prepared movement is executed, the rule should wait for Processor Free. Waiting for Modality Free is the slowest and most conservative approach, but it may be optimum if the feature programming for the next movement in fact depends on some result of the previous movement. Currently it is unclear whether the Execution Free state has any value.

Some of the motor processor styles also generate separate signal clauses for the cognitive processor. These clauses are of the form:

(Motor <modality> <signal description>)

The contents of the signal description depend on the processor and the style, but generally consist of a movement description based on the features followed by state label. Thus these signals correspond to "efferent copy feedback" about the movement. For example, a Manual Punch movement might result in the following clause being added to production system memory at a certain time:

(Motor Manual Punch Right Index Finished)

These clauses are deleted from the cognitive processor production system memory after the time given by the parameter Efferent_deletion_delay. At this time, the movement signals are somewhat redundant with the Busy/Free state signals, but they do contain more information about the exact status of the movement execution, rather than just the state of the processor.

Aimed movements

Aimed motions to objects in space typically involve computing the direction and extent of the movement from the current location to the target location. These features are expressed as polar vector (r, θ) giving the distance and angle of the target location from the current location of the effector.

Commands to make a motion to an object in physical space refer to psychological visual objects; the current location of the object in sensory storage is used to determine the destination of the movement; this location might be noisy or might lag behind the object's true position.

Current Parameters

Feature_time: 50 ms

Initiation_time: 50 ms

Burst_time: 100 ms. This parameter is used to define a minimum possible physical execution time.

Efferent_deletion_delay: 75 ms

Execution_fluctuation_factor: mean 1.0, sd .5, Normal distribution. This parameter can be used to produce variability in movement times.

Movement symbols, movement events, and named movements

Rationale

The EPIC software includes some programming conveniences that are related to some important theoretical constraints and assumptions. Fundamentally, the cognitive processor is supposed to instruct the motor processors only in terms of symbols for movements that the motor processor knows how to translate into actual movements. In addition, the device in the task environment module has to know which movements correspond to what inputs from the simulated human. The problem for the model-builder is that the motor processor code has to contain all symbols and their translations into both features and device inputs before the model can be run. For convenience in writing models, some facilities are available for defining such symbols and their translations.

Device inputs versus movement features

Some movement styles need to produce a particular Symbol as input to the simulated device, but also specify the particulars of how the movement is to be produced. For example, pressing a button means that the device has to be informed of which button is being pressed, but also the manual motor processor needs to be commanded as to which hand and finger should be used. The approach taken in the current EPIC is that in such movement commands, both the device symbol and the movement features are specified.

For programming convenience, some movement styles assume a conventional name for an operated-upon part of the device. For example, the Point command refers only to the target for the motion; the cursor (the visible object whose position is being manipulated) is assumed to be the object whose physical name is "Cursor". In contrast, the Ply command can be used for the same purpose, but it requires two psychological object names - one is the cursor object, the other is the target object. In order to enable the device to get inputs identified in its own terms, the psychological name for the target is replaced by the physical name for the target object in the event sent to the device. This is done by the movement execution code consulting the physical/psychological name table maintained by the visual system to find the physical name to send to the device.

Named locations

Since the cognitive processor has no direct access to visual space location, or any way to create such location information, there has to be some provision for defining locations in visual space, so that the eyes can be directed to a particular point on a display even when there is no object there to look at. This is done with a named location, which is a sort of pseudo-object that has a location, but no other properties. Currently, named locations can only be used for eye movements, though in principle they could be applied to other kinds of aimed movements.

For example, (Define Named_location Fixation_point -24.0 0.0) assigns the name Fixation_point to a location at (-24.0, 0.0) in visual space.

The psychological aspect of this mechanism is that humans can learn that certain visual events might occur at certain otherwise undistinguished places (e.g. a certain point on a blank computer display). However, such

learning would presumably take some time, and named locations should not be used if the modeler believes that a true visual search for an object is necessary.

Manual_processor and Manual_actions

Overview

The manual processor controls the hands and fingers, and there are several different movement styles represented in manual actions. The complete set in the LISP version of EPIC is not yet fully available in current EPIC, but these previous movement styles can be quickly added as needed. Some styles are very simple; for example, the Punch style, used for pressing a button that is directly beneath a fingertip, has features specifying just the the hand and finger involved. Several styles are more complex, involving the direction and extent of a movement, such as the Ply style, used to move a cursor on to a target (e.g. with a joystick) in single aimed movement. Some of the styles, such as Keystroke, are experimental attempts to supply the useful approximations from the GOMS Keystroke-Level Model to potentially simplify modeling where more exact results are not required.

Punch

Purpose:

The Punch style is used for high-speed responses where the button is assumed to be directly under a finger, as in typical human performance experiments.

Command syntax:

(Send_to_motor Manual <command> Punch <device button name> <hand> <finger>)

The <command> is either Prepare or Perform. <hand> must be either Right or Left. <finger> must be one of Thumb, Index, Middle, Ring, or Little. Punch involves no hand motion. Fingers are simply downstroked, then upstroked. The symbol provided as the <device button name> (either a variable or a constant) is sent to the device.

Feature preparation:

The features are Style: Punch, Hand: <hand>, Finger: <finger>.

The feature hierarchy is Style > Hand > Finger: changing the Hand requires recomputing the Finger feature.

The movement description for signals is (Motor Manual Punch <hand> <finger> <state>)
Where <state> is Started or Finished.

Movement execution after initiation:

Schedule a Cognitive_Add_Clause_event to immediately arrive at the Cognitive_processor, containing the Started signal; schedule a Cognitive_Delete_Clause_event to remove the signal after Efferent_deletion_delay.

Schedule a Device_Keystroke_event containing the <device button name> to arrive at the device after an additional time given by key_closure_time of 25 ms * Execution_fluctuation_factor.

After additional time Burst_time after initiation, the downstroke is complete:
Schedule a Cognitive_Add_Clause_event to immediately arrive at the Cognitive_processor, containing the Finished signal; schedule a Cognitive_Delete_Clause_event to remove the signal after Efferent_deletion_delay.

After an additional time Burst_time after initiation, the upstroke is complete, and the movement is complete at this time.

Ply

Purpose:

The Ply style is used for movements in which the simulated human manipulates a control, such as a joystick or mouse operated by either hand, to cause one visual object, the *cursor*, to move to the same position as another visual object, the *target*. If a standard computer mouse is involved, the Point style is generally more convenient. The device is sent a series of events to crudely simulate the movement of the cursor in a linear fashion; the last of these events arrives at the end of the movement time. Thus, the device is sent a serious of events, allowing it to "animate" the cursor motion, rather than having the device see only a single new position (a "jump") at the end of the movement time. Note that these subevents are not psychomotor submovements, and do not represent actual behavioral movement velocity.

Command syntax:

(Send_to_motor Manual <command> Ply <cursor object> <target object> <hand>)

The <command> is either Prepare or Perform. The <cursor object> and <target object> must be currently visible visual objects. <hand> must be either Right or Left. The <hand> is assumed to be on the joystick or manipulandum.

Feature preparation:

The features are: Hand: <hand>, Direction: polar vector angle between cursor and target, Extent: polar vector length between cursor and target. The polar vector is computed during preparation.

The feature hierarchy is Style > Hand > (Direction = Extent): changing the Hand requires recomputing both Direction and Extent, but Direction and Extent are independent of each other and can be changed separately.

The Extent feature is considered reusable if the difference between the new and previous extent is less than 2 degrees of visual angle.

The Direction feature is considered reusable if the difference between the new and previous direction is less than $\pi/4$ (.785) radians.

The movement description for signals is (Motor Manual Ply <hand> <state>)

Where <state> is Started or Finished.

Movement execution after initiation:

Schedule a Cognitive_Add_Clause_event to immediately arrive at the Cognitive_processor, containing the Started signal; schedule a Cognitive_Delete_Clause_event to remove the signal after Efferent_deletion_delay.

Compute the target distance and target size.

The distance is the center-to-center distance from the cursor to the target, using the locations at the time of movement preparation. The target size is defined as the length of the line segment extending from the cursor center through the target center that is enclosed in a rectangular bounding box centered on the target whose dimensions are the target's visual Size property. This does not take the actual shape of the target into account, but does take the direction of approach into account, and is exact if the target is rectangular.

Compute the movement time using a modification of Welford's form Fitts' Law:

ID = (target distance / target size) + 0.5

movement time = 100 ms * log_base_2(ID)

if ID is < 1, or the calculated movement time is less than 100 ms, then use 100 ms for the movement time.

multiply the movement time by Execution_fluctuation_factor.

Compute a number of subevents to be sent to the device, as

number of subevents = total movement time / 25 ms

compute the distance for each subevent as total distance / number of subevents

Schedule a Device_Ply_event to the device for exact subevent to arrive at each subevent time, giving the new position of the cursor along the movement vector, the subevent distance for each one. These intermediate events carry the physical name of the cursor object, but a target name of "nil".

Schedule a final Device_Ply_event that gives the target location as the final location, and contains the physical name of the cursor object and the target object.

Schedule a Cognitive_Add_Clause_event to immediately arrive at the Cognitive_processor, containing the Finished signal; schedule a Cognitive_Delete_Clause_event to remove the signal after Efferent_deletion_delay.

The movement is considered complete at the final Device_Ply_event time.

Possible extensions: In the future the Ply style might allow for different control-order characteristics, but at this time, it implements only a Fitts' Law movement. The literature is sparse on the subject of details of control movements involved with different control-order devices.

Point

Purpose:

In many human-computer interaction situations, a computer mouse is used to move the cursor, and this is the

only pointing device available, and it is normally in the right hand. To simplify modeling in this domain, the Point style was defined - it is very similar to the Ply style, sharing most of the implementation with a general Manual_aimed_action class.

To use the Point style successfully, the Device must use "Cursor" as the physical name of the cursor object; this is recognized by the Eye_processor, and "Cursor" is also assigned as the psychological name for the object. The device must likewise recognize that a Device_Ply_event that names "Cursor" is in fact referring to the mouse cursor.

Another feature of Point is that if the Size of the target has not been specified and has defaulted to [0, 0], the times will default to those corresponding to the GOMS Keystroke-Level Model Pointing operator.

Command syntax:

(Send_to_motor Manual <command> Point <target object>)

The <command> is either Prepare or Perform. The <target object> must be a currently visible visual object. The Right hand is assumed to be on the mouse.

Feature preparation:

If the target size is the default of [0, 0], two features are assumed to be prepared. Otherwise, the feature preparation is identical to the Ply style.

The movement description for signals is (Motor Manual Point <hand> <state>)
Where <state> is Started or Finished.

Movement execution after initiation:

If the target size is the default of [0, 0] the movement time is computed so that the total of feature preparation, initiation, and movement is the Keystroke-Level GOMS Model time for pointing, specified as the parameter KLM_point_time. Otherwise, the execution time is identical to the Ply style.

The movement execution process is identical to the Ply style.

Future extensions: The coefficient for the Fitts' Law movement will be contingent on whether a mouse button is being held down or not; this will require defining some standard names for mouse buttons.

Keystroke (experimental)

Purpose:

The Keystroke style is as a rough approximation, as in the GOMS Keystroke-Level Model, for hitting a key anywhere on the keyboard. Implicit is some kind of direction and extent features, roughly approximated as a total of three features, of which half are shared with a previous movement of the same style.

Command syntax:

(Send_to_motor Manual <command> Keystroke <device key name>)

The <command> is either Prepare or Perform.

Feature preparation:

If the previous movement was a Keystroke to the same <device key name>, no preparation time is required. If the previous movement was a Keystroke to a different key, 1.5 features must be prepared. Otherwise, 3 features must be prepared.

No signals are generated.

Movement execution:

In the future, the position of the hand on either the mouse or keyboard will be tracked, and if the hand is not on the keyboard, the time for a Keystroke-Level GOMS Model "Home" operation, which requires the specified time, Homing_time will be added. No preparation is required for the home operation.

Schedule a Device_Keystroke_event containing the <device button name> to arrive at the device after the movement time, which is computed so that the total of feature preparation, initiation, and movement is the Keystroke-Level GOMS Model time for pointing, specified as the parameter KLM_keystroke_execution_time.

The movement is complete at this time.

Current Parameters

Key_closure_time: 25 ms

Ply_minimum_time: 100 ms

Ply_coefficient: 75 ms

Subevent_time: 25 ms

Point_coefficient: 100 ms

Point_coefficient_button_held: 135 ms

Homing_time: 400 ms

KLM_keystroke_execution_time: 280 ms

KLM_point_time: 1100 ms

Future Extensions to Manual_actions

Additional manual styles were defined in the LISP version of EPIC, and some of these need to be added for completeness, and a few extensions made, such as holding and releasing buttons, as on a mouse. If it appears to be useful, additional manual action styles based on the Keystroke-Level GOMS Model can be quickly added.

The set of Keystroke-Level GOMS Model actions needs to be fully implemented in terms of automatically keeping track of whether the hand is on the mouse or the keyboard, and automatically generating the required homing movements when required. Policy will have to be defined for whether these automatically generated movements apply only between the Keystroke-Level GOMS Model movements.

Ocular_processor, Invol_ocular_processor, and Ocular_actions

Overview

There are two separate, but related, motor processors that move the eye. The Ocular_processor receives commands from cognitive processor production rule actions, and so produces voluntary saccadic eye movements. The Invol_ocular_processor receives information about visual events from the visual system, and produces reflexive and centering eye movements, both saccadic and smooth movements. It can be controlled by production rule actions only to the extent of enabling or disabling the centering and reflex *modes,* but the cognitive processor can not control the involuntary movements directly at all. Only the voluntary ocular processor can be sent commands by production rules; control over the involuntary processor modes is handled by commands made to the Ocular processor. In other words, the Involuntary ocular processor is "hidden" from cognition.

The two processors can prepare movements independently of each other, but the voluntary system takes precedence over the involuntary one. That is, voluntary movements are given priority in that once prepared, they are always executed instead of any involuntary movements that are also ready to execute; such preempted prepared involuntary movements are simply discarded and will not be executed. However, once a physical movement starts execution, it is always allowed to complete regardless of whether it was voluntary or involuntary in origin.

Similar to the manual processor, the ocular processor has an experimental rough-approximation command based on GOMS Keystroke-Level Model concepts.

Voluntary system: Ocular_processor

In the LISP version of EPIC, there were several different movement commands because setting the mode of the involuntary system as part of the eye movement is often convenient. For example, the FIXATE command caused a saccade to a location, but also turns off centering and reflex movements so that the eye will stay fixated on the specified location regardless of other visual events. Currently, only the most commonly used eye movement command is implemented, the Move command, that turns on centering and turns off reflex movements at the end of the saccade.

Move

Purpose:

The Move command is the basic command used to produce an eye movement to a specified location, given either by naming a visual object, or a named location.

Command syntax:

(Send_to_motor Ocular <command> Move <visual object>)

The <command> is either Prepare or Perform. The <visual object> must be a currently visible visual object or the name of a named location.

Feature preparation:

The features are: Direction: polar vector angle between current eye position and target, Extent: polar vector length between current eye position and target. The polar vector is computed during preparation.

The feature hierarchy is Style > (Direction = Extent): Direction and Extent are independent of each other and can be changed separately.

The Extent feature is considered reusable if the difference between the new and previous extent is less than 2 degrees of visual angle.

The Direction feature is considered reusable if the difference between the new and previous direction is less than $\pi/4$ (.785) radians.

If the eye is already in the target position (Extent < 0.1 degrees) , the preparation time is zero, and the eye is considered to already be in position.

Movement execution after initiation:

If the eye is already in position, the execution time is zero and no events are sent to the Eye_processor.

Compute the eye movement time using Ocular_processor and Motor_processor parameters as
(Voluntary_saccade_intercept + Voluntary_saccade_slope * distance) * Execution_fluctuation_factor

Schedule an Eye_Voluntary_Saccade_Start_event to immediately arrive at the Eye_processor, containing the

future position of the eye and the movement completion time.

Schedule a Device_Eyemovement_Start_event that contains the same information to arrive at the device at the device immediately. These device events allow the simple simulation of eye movement experiments.

Schedule an Eye_Voluntary_Saccade_End_event to arrive at the Eye_processor after the eye movement time has elapsed, containing the flags to set centering on and reflexes off. See the Eye_processor and Invol_ocular_processor descriptions for information about centering and reflex movements.

Schedule a Device_Eyemovement_End_event that contains the same information as the Device_Eyemovement_Start_event to arrive at the device at the eye movement end time.

The movement is considered complete at the eye movement end time.

Set_mode

Purpose

This is not really a movement style, but rather a specialized motor "command" to allow cognitive processor production rules to control the *modes* in which the Eye_processor and the Invol_ocular_processor carry out involuntary actions. Here is specified the preparation and execution time of these commands (purely speculative), while the section on the Involuntary system describes the modes themselves.

Command syntax:

(Send_to_motor Ocular Perform Set_mode <command> <mode name>)

The <command> is either Enable or Disable. The <mode name> is either Centering or Reflex.

Feature preparation:

Two features are assumed to be required.

Execution after initiation:

The Centering or Reflex mode is set to Enable or Disable, and the action is considered complete after Burst_time.

Involuntary system: Invol_ocular_processor

The involuntary ocular processor works together with Eye_processor to implement involuntary eye movements, if commands sent by the cognitive processor have enabled the involuntary eye movements. Basically, the Eye_processor detects the need for an involuntary eye movement, such as as object needing to be centered on the fovea, and sends a command to the Involuntary ocular processor. The involuntary ocular processor prepares and executes an eye movement, which causes eye movement events to be sent to the Eye_processor.

There are two involuntary eye movement mechanisms. One is the simple servo-like *Centering* system that attempts to keep a visual object centered on the foveal bouquet. Centering implements "automatic" tracking or zeroing-in motions without cognitive involvement (once enabled). At this time, the centering system works only in terms of which visual object is closest to the center of the fovea; it does not deal with the object name or identity at all. Thus it is possible for the centering to go astray if the wrong object happens to be closer to the foveal bouquet. The centering system produces either a smooth movement for small distances or a saccade for larger distances to put the nearest object in the center of the fovea. If Center is enabled, the Eye_processor determines which type of movement is required, and to what object, and sends the command to Invol_ocular_processor. After the movement is complete, the Eye_processor may issue an additional movement command.

The second type of involuntary eye movement mechanism is the *Reflex* system. If Reflexes are enabled and on, and the Eye_processor detects certain types of events, such as a new object appearing, or an object moving, it sends a command to Invol_ocular_processor, which prepares and executes a saccade to the object.

Whether centering or reflex motions will be made is determined not only by whether the cognitive processor has permitted them, but also by whether the visual events are appropriate. For example, reflex movements are appropriate to make if there are currently no objects to center upon, but only if they have been enabled. Once a reflex movement has been made, no more will be made until either a cognitive processor command or appropriate visual situation dictates that a new reflex movement will be appropriate. Thus, involuntary movements are controlled by two state flags in Eye_processor, an enablement flag controlled only by cognitive processor production rule commands, and an On/Off flag, controlled either by cognitive commands or by the visual situation. See the section on Eye_processor for

specifics on when a command will be issued to Invol_ocular_processor.

The following commands are thus "internal" - they cannot be issued by production rule actions, and so have no "syntax" to describe.

Invol_ocular_Saccade_action

Purpose:

This command is issued by Eye_processor in order to produce an involuntary saccade to a specified object.

Feature preparation:

The preparation time for an involuntary saccade is assumed to be identical to that of a voluntary saccade.

Movement execution after initiation:

The movement execution time is assumed to be determined in a way identical to that of an involuntary saccade, but has its own parameters, Involuntary_saccade_intercept and Involuntary_saccade_slope. An Eye_Involuntary_Saccade_Start_event and an Eye_Involuntary_Saccade_End_event are sent to Eye_processor, and these contain no information about changing the Centering or Reflex modes. Also, in the current version, no events are sent to the device, on the assumption that the modeler is probably not interested in modeling eye movement experiments that involve involuntary saccades. Finally, an additional Involuntary_saccade_refactory_period must elapse before the movement is considered complete.

Invol_ocular_Smooth_move_action

Purpose:

This command is issued by Eye_processor in order to produce a smooth movement to a specified object.

Feature preparation:

This movement is assumed to require no preparation time, so it is potentially very fast.

Movement execution after initiation:

If the eye is already in position (movement Extent < 0.1 degrees) no movement is made and the movement is considered complete after initiation time.

Compute the movement time as
(Involuntary_saccade_intercept + Involuntary_saccade_slope * distance) * Execution_fluctuation_factor

An Eye_Involuntary_Smooth_Move_Start_event is sent to the Eye_processor, followed by an Eye_Involuntary_Smooth_Move_End_event after the movement time. In the current version, no events are sent to the device.

The movement is considered complete after an additional Involuntary_smooth_move_refractory_period has elapsed.

Current Parameters

Voluntary_saccade_intercept: 0 ms

Voluntary_saccade_slope: 4 ms

Involuntary_saccade_intercept: 0 ms

Involuntary_saccade_slope: 4 ms

Involuntary_saccade_refactory_period: 10 ms

Involuntary_smooth_move_intercept: 0 ms

Involuntary_smooth_move_slope: 2 ms

Involuntary_smooth_move_refractory_period: 10 ms

Future Extensions to Ocular_actions

The LISP EPIC included some additional voluntary eye movement commands that differed only in terms of the Centering and Reflex modes - such as Fixate, which turned Centering and Reflexes off, and Preposition, which turned

Reflexes on. These will need to be added to the current version to enable modeling of a larger variety of eye movement experiments. Also, the onset of a sound was able to trigger an reflexive eye movement to the location of the sound source, similar to an onset of a visual object. This linkage between the auditory and ocular system needs to be added to the current version.

The Vocal_processor and Vocal_actions

Overview

The vocal motor processor produces speech output. At this time, it is a very simple mechanism with limited possibilities because the models to date have had very simple requirements for vocal processing.

Movements may be prepared in advance, and preparation and execution can be overlapped, but the psychological realism of these possibilities is currently unknown. A highly simplified concept for extended speech is currently in place that incorporates the rough-and-ready idea that once speech is underway, the preparation for future movements is fully overlapped with the current execution. The time taken to speak an utterance is assumed to be linear with the number of syllables.

Speak

Purpose:

The Speak style produces utterances that must be least one word in length.

Command syntax:

(Send_to_motor Vocal <command> Speak <word> ...)

The <command> is either Prepare or Perform. At least one <word> must appear; any additional number is accepted, and make up the complete utterance. These are written as single whitespace-delimited strings or variable names.

Feature preparation:

If the prevous commanded movement style is different from Speak, three features are assumed. If the previous movement is Speak, and the utterance is identical to the previous utterance, then zero features have to be prepared; otherwise, two features are assumed. Note that this number does not depend on the actual content of the utterance, nor on the length of the utterance.

The movement description for signals is (Motor Vocal Speak <utterance> <state>)
Where <state> is Started or Finished.

Movement execution after initiation:

Compute the speech duration as
duration = (number of syllables in the utterance) * Syllable_time * Execution_fluctuation_factor
where the number of syllables is computed using a crude algorithm pending an implementation based on actual English morphology and phonemics; The algorithm basically defines syllables as the number of vowel clusters in a word, with some special cases represented. Also, based on previous modeling work, the algorithm is somewhat "smart" about acronyms and numbers.

Compute the movement completion time as the current time + duration.

Schedule a Cognitive_Add_Clause_event to immediately arrive at the Cognitive_processor, containing the Started signal; schedule a Cognitive_Delete_Clause_event to remove the signal after Efferent_deletion_delay.

Schedule a Device_Vocal_event containing the utterance and the duration to arrive at the device immediately, and

Schedule a Cognitive_Add_Clause_event to arrive at the Cognitive_processor at the completion time, containing the Finished signal; schedule a Cognitive_Delete_Clause_event to remove the signal after Efferent_deletion_delay.

The movement is complete at the movement completion time.

Current Parameters

Syllable_time: 150 ms

Future Extensions to Vocal_processor

To support modeling of verbal working memory, the LISP EPIC included both overt and covert styles of speech, with a connection to auditory memory such that vocal output would get automatically deposited into auditory working

memory, accompanied by additional information that made is possible to represent individual words (as in a verbal working memory experiment) and note which had prosodic indicators as forming the beginning or end of a string of such utterances. These mechanisms will have to be added to the current version to support the same models of verbal working memory.

References

Bovair, S., Kieras, D. E., & Polson, P. G. (1990). The acquisition and performance of text editing skill: A cognitive complexity analysis. *Human-Computer Interaction*, 5, 1-48.

Bregman, A. S. (1990). *Auditory scene analysis: The perceptual organization of sound*. Cambridge, MA: MIT Press.

Covrigaru, A., & Kieras, D. E. (1987). PPS: A Parsimonious Production System (Tech. Rep. No. 26). (TR-87/ONR-26). Ann Arbor: University of Michigan, Technical Communication Program. (DTIC AD A182366).

Findlay, J.M., & Gilchrist, I.D. (2003). *Active Vision*. Oxford: Oxford University Press.

Forgy, C. (1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19, 17-37.

Kieras, D.E., Meyer, D.E., Mueller, S., & Seymour, T. (1999). Insights into working memory from the perspective of the EPIC architecture for modeling skilled perceptual-motor and cognitive human performance. In A. Miyake and P. Shah (Eds.), *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control*. New York: Cambridge University Press. 183-223.