The Control of Cognition[1]

David Kieras

University of Michigan

A preprint of:

Kieras, D.E. (in press). The control of cognition. In W. Gray(Ed.), *Integrated models of cognitive systems*. Oxford University Press.

Introduction

Where we were

Inspired by the computer metaphor, a few decades ago psychologists were describing the mind with "box models" like those illustrated in Figure 1 (e.g. see Bower & Hilgard, 1981, ch 13; Norman, 1970). Such models were supposed to illustrate the flow of information between processing stages or storage mechanisms. Sensory and motor organs were not in the picture, or if they were, they were not represented in any detail. Since most of the experiments addressed by the models involved memory paradigms, the storage systems were emphasized, but the contents of them were collections of items loosely-specified in terms of "features." The processes by which information moved between the storage systems was not elaborated in any detail, and the "executive" was somehow responsible for overseeing all of the processing and changing it to suit the needs of the task. To the extent there was rigor in these models, it was supplied by simple probability-theory models for the individual boxes, such as the probability that an item would be lost as a function of time, or information was successfully transferred from one box to another.

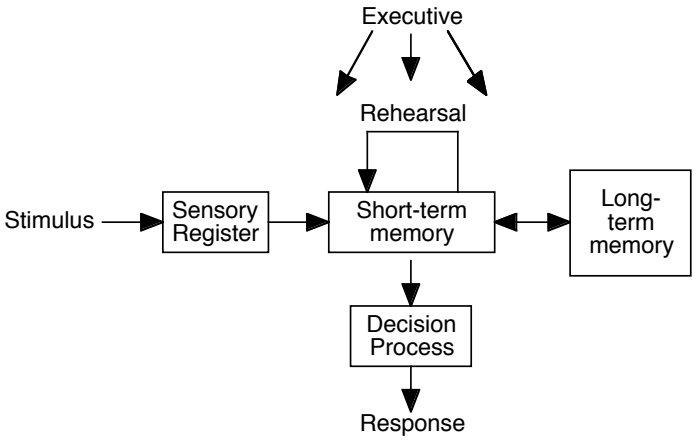

Figure 1. A typical "box model" in human information-processing theory.

[1] Work on this paper was supported by the Cognitive Science Program of the Office of Naval Research, under grant number N00014-03-1-0009 to David Kieras and David Meyer. David Meyer's contribution to these ideas has been substantial and essential, as indicated by the many co-authored papers cited in this chapter.

Where we are

At the current time, many psychologists are describing the mind with models built within a cognitive architecture whose top-level structure is illustrated also with a box diagram such as Figure 2, which illustrates the EPIC architecture that will be described in somewhat more detail later in this chapter. But there are many critical differences between what is implied by these current diagrams and the earlier "box models." The sensory-motor peripherals are in the diagram, reflecting how their properties are an explicit part of the model. Both memory systems and processing systems are shown, corresponding to the concept that a psychological theory must include both how knowledge is represented and how it is processed. There is no mysterious "executive" because all control of activity is done by the cognitive processor, which in addition to performing cognition, also controls the system as a whole, including itself. Thus the cognitive controller is synonymous with the cognitive processor – there is no other component in the system that controls the overall activity. Thus, for the rest of this chapter, the cognitive controller of the chapter title is identified with the cognitive processor in a cognitive architecture.

Figure 2's content is an example of a cognitive architecture in that the diagram represents an hypothetical collection of fixed mechanisms, analogous to a computer's general-purpose hardware, that are "programmed" by information in memory with the knowledge of how to perform a specific task, analogous to a particular piece of software that might be running on a computer. A complete model for performance in a particular task thus consists of the hypothetical task-independent fixed architecture together with the task-specific knowledge in memory for how to perform the task. This attempt to partition psychological models into task-independent and task-dependent portions is the hallmark of current cognitive theory.

Overview of the Chapter

The subject of this chapter is the cognitive processor and how it controls the architecture as a whole, and itself as well. The basic control functions are to control input into the system, output from the system, and the control of cognition in terms of basic sequential processing, parallel processing, interrupt handling, and executive processes that monitor and control cognitive activity. The presentation is strictly theoretical (or meta-theoretical); no data are

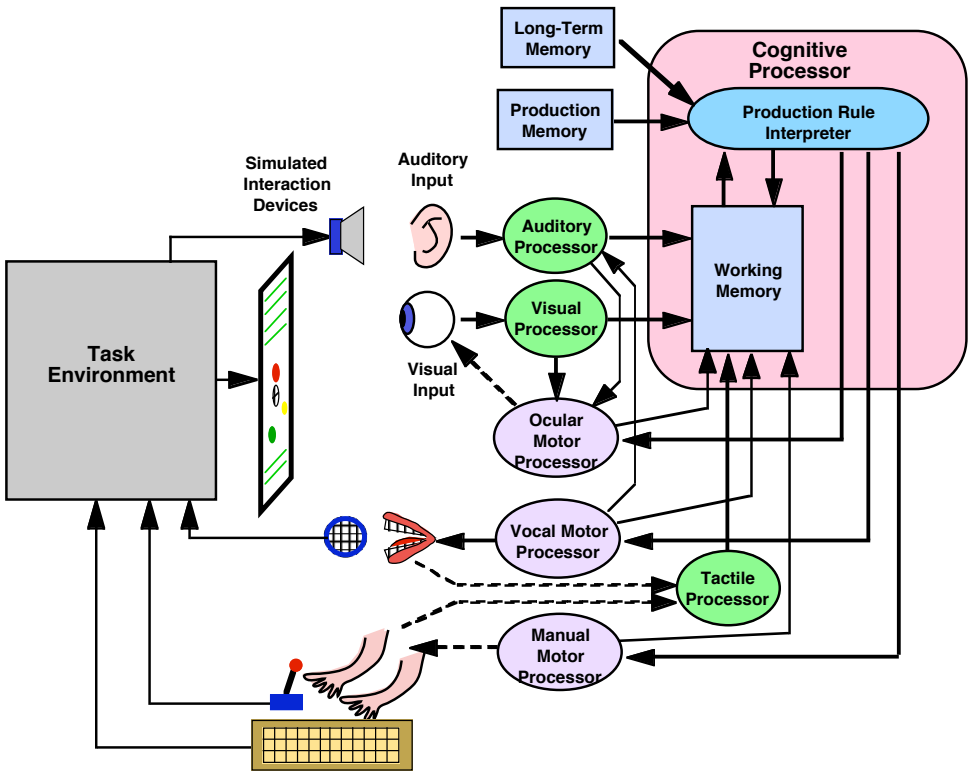

Figure 2. The EPIC architecture in simplified form. The simulated environment, or device, is on the left; the simulated human on the right.

presented or compared to any models. The EPIC architecture will be used as the basic framework for discussion, because it illustrates many of the issues well. Examples will be drawn from actual working EPIC models.

The discussion will be based on the computer metaphor, which historically underlies the development of information-processing psychology, but it is actually more useful than ever. That is, the computer system was an obvious metaphor for human information-processing activity, and thus helped usher in a post-behavioristic "mentalism" that made it possible to discuss mental activity in a rigorous scientific fashion. However, the original computer metaphor was based on early computer technology circa 1960, and it was not updated as computers evolved. Current computer systems (see Tucker, 2004) are much more sophisticated, and have been so for a long time: they perform simultaneous I/O and computation, parallel and concurrent processing of multiple tasks, and are controlled by a general-purpose "executive" – an Operating System (OS). Thus the modern computer system is a valuable metaphor for cognitive control processes in humans because the corresponding psychological theory of control and executive processing is poorly developed and vague; in contrast, these are very well developed topics in computer technology.

The obvious objection to the computer metaphor is that the brain is not a computer! The "hardware" of the brain is fundamentally different, and is vastly more complex and sophisticated. However, the computer system is the only information processing system on the planet that has fully understood control and executive mechanisms. The concepts involved both provide a working concrete example of such mechanisms, and set a lower bound on the complexity and sophistication of mechanisms that should be considered by psychological theory for similar functions. Thus applying the modern computer metaphor should be useful even if we have profound reservations about its applicability in detail. The relevant computer-metaphor concepts will be introduced as needed in the discussion.

First, the EPIC architecture will be summarized. Then how the cognitive processor controls input and output will then be presented, followed by the bulk of the chapter, on the control of cognition. This is divided into two sections: The first covers the basic control of cognition, which concerns sequential, hierarchical, parallel, and interrupt flow of control. The second discusses executive control of cognition. Executive processes are illustrated with an example of two different executive regimes for a dual-task situation.

Description of the EPIC Architecture

Basic structure

Figure 2 shows the overall structure of the EPIC architecture. The cognitive processor consists of a production rule interpreter that uses the contents of production memory, long-term memory, and the current contents of a *production-system working memory* to choose production rules to fire; the actions of these rules modify the contents of working memory, or instruct motor processors to carry out movements. Auditory, visual, and tactile processors deposit information about the current perceptual situation into working memory; the motor processors also deposit information about their current states into working memory. The motor processors control the hands, speech mechanisms, and eye movements. All of the processors run in parallel with each other. Note that the *task environment* (also called the *simulated device*, or simply the *device*) is a separate module that runs in parallel as well. The pervasive parallelism across perception, cognition, and action motivated the design of EPIC and is reflected in the acronym: **E**xecutive **P**rocesses **I**nteract with and **C**ontrol the rest of the system by monitoring their states and activity. Because of this fundamental focus, EPIC is a good choice for a discussion of cognitive control mechanisms.

Figure 2 is a simplified view; each of the processors is fairly complex, as illustrated by Figure 3 that shows additional detail related to the visual system. The environment controls the current contents of the *physical store*; changes are sent to the *eye processor*, which represents the retinal system and how the visual properties of objects in the physical store are differentially available depending on their physical properties such as color and size, and their eccentricity – the distance from the center of the fovea. The resulting "filtered" information is sent to the *sensory store*, where it can persist for a fairly short time, and comprises the input to the *perceptual processor*, which performs the processes of recognition and encoding. The output of the perceptual processor is stored in the *perceptual store*, which is the *visual working memory*, and whose contents make up the visual modality-specific partition of the production-system working memory. Thus production rules can test visual information only in terms of the current contents of the *perceptual store*. Production rules can send commands to *involuntary* and *voluntary ocular processors* which control the position of the eyes. The voluntary ocular processor is directly controlled by the *cognitive processor*, which can command an eye movement to a designated object whose representation is in the perceptual store. The involuntary ocular processor generates "automatic" eye movements, such as reflex saccades to

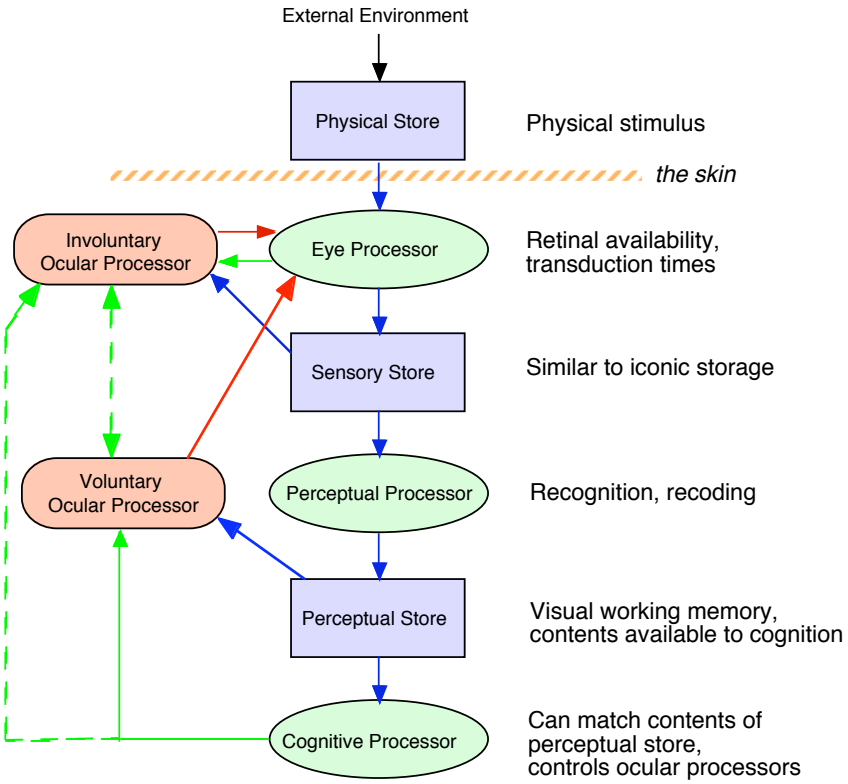

Figure 3. The internal structure of EPIC's visual processor system. This is an expansion of the visual processor module shown in Figure 2.

a new or moving object, or smooth movements to keep the eye foveated on a slowly moving object, using the location information present in the sensory store. The cognitive processor can only enable or disable these automatic activities and so only indirectly controls the involuntary processor. More detail can be found in Meyer & Kieras (1997a), Kieras & Meyer (1997), and Kieras (2004).

Production Rules

The cognitive processor uses a very simple production system interpreter called the *Parsimonious Production System* (PPS. This interpreter has very simple syntax and semantics, but is implemented with a full rete-match algorithm (Forgy, 1982), so that it scales well in terms in of performance to large sets of production rules. The items in the production system working memory are simply lists of symbols – the interpreter does not assume any particular structure of memory; rather these are defined in terms of what inputs to working memory are provided by the perceptual and motor processors, and programming conventions for the production rules themselves.

Like most production systems, PPS operates in cycles, which have a 50 ms period. During a period, the perceptual and motor processors continually update the production system working memory. At the beginning of each cycle, the additions and deletions of working memory items specified by the last rule firing are carried out, and then all production rules whose conditions are currently met are "fired" – their actions are executed. Note that *all* rules currently satisfied are fired on each cycle. Also, if there is more than one specific instantiation of a rule's conditions present in working memory, the rule will be fired for each of them in the same cycle. There is no preset limit on how many rules can fire, or how many instantiations of them can fire, on a single cycle.

This full parallelism is often mistaken for a claim of unlimited processing power, but it is not. The peripheral mechanisms limit how much information can be simultaneously present from perceptual and motor sources, and some limits are normally observed for the amount of information in working memory created and maintained by the production rules, but this is not enforced by the architecture. Note that as argued by Kieras, Meyer, Mueller & Seymour (1999) the true limits of working memory are not at all well understood empirically – e.g. compare

Baddeley & Logie (1999) with Ericsson & Kintsch (1995; Ericsson & Delaney, 1999). Working memory is either the traditional small-capacity modality-specific "scratchpad" memory, or it is a huge-capacity fast-write usage of long-term memory enabled by well-learned task-specific retrieval strategies. These two concepts are probably compatible, but at this date there is not a theoretical consensus on what the "real" working memory is like. So the fact that EPIC does not impose working memory limits is a tactical decision on our part not to "hard-wire" our collective ignorance into the architecture, but rather to let experience with modeling various tasks determine what kinds and amounts of information are required for plausible modeling of important phenomena, and then use this knowledge to constructively specify and explain memory limitations and their sources. For example, the work in Kieras et al. (1999) suggests that verbal working memory limitations have their basis not in a direct limit on how many items can be kept active in memory, but rather is due to how long phonological representations persist in auditory memory relative to the speed with which they can be refreshed through rehearsal (see also Mueller, Seymour, Kieras, & Meyer, 2003), and how well the recall strategies can recover information from the resulting mélange of partially-decayed representations (Mueller, 2002).

The partitions of the production-system working memory are one for each perceptual modality (e.g. auditory and visual), which are synonymous with the perceptual stores for the modality, and another partition for the states of the motor processors. A *Control Store* partition contains *Goal*, *Step*, and other similar items that are used to specify the flow of control through production rules. The *Tag Store* partition contains *Tag* items which simply associate a item in a modality working memory with a symbol designating a role in the production rules, analogous to a variable and its binding in a traditional programming language. That is, one set of production rules might be responsible for identifying which visual object is "the stimulus" for a task, and add a Tag accordingly. Other production rules could then refer to the current stimulus using the Tag instead of having to re-identify the stimulus repeatedly.

Figure 4 shows two sample production rules from working models to illustrate how PPS rules look. Various aspects of these rules will be explained later, but for now it suffices to provide a simple overview of the basic syntax. The first rule identifies the visual object for a fixation point consisting of a red cross-hair. The second rule moves the cursor to a visual object. Each rule begins with a rule name, then a set of conditions after the *If*. The conditions are implicitly a conjunction; all of them must be satisfied for the rule to be fired. The actions listed after the *Then* are executed if the rule is fired. The *Add* and *Delete* actions modify the contents of the production system working memory. *Send_to_motor* is a command to a motor processor. Terms starting with a *?* character are *variables*. In a condition, the variables are assigned whatever values necessary to make the condition item match an item in working memory. Variables appearing in actions take on the values assigned to them in the condition when the action is executed. Note that the variables are scoped only within the rule; once the rule is fired and executed, the variable values are discarded; there is no memory of them from one rule to the next, or one cycle to the next.

Control of Input

In computers, the process of input is under the ultimate control of the *Central Processing Unit* (CPU). That is, input from peripheral devices is normally only brought into the main memory if the CPU has instructed it. In early machines, instructions executed by the CPU performed the input/output (I/O) processing, but in modern machines (since about 1970) a hardware subsystem does all of the detail work. Basically, the CPU executes an instruction that commands an input subsystem to deliver specified information to an area of memory. Thus input involves a

```
(Top-see-fixation-point
If ((Goal Do Visual_search) (Step Waitfor Fixation-present)
   (Visual ?object Shape Cross_Hairs) (Visual ?object Color Red))
Then ((Add (Tag ?object fixation-point))
   (Delete (Step Waitfor Fixation-present)) (Add (Step Waitfor probe-present))))

(Top-make-response
If ((Goal Do Visual_search) (Step Make Response)
   (Tag ?target target) (Tag ?cursor cursor)
   (Motor Manual Modality Free))
Then ((Send_to_motor Manual Perform Ply ?cursor ?target Right)
   (Delete (Step Make Response)) (Add (Step Make Response2))))
```

Figure 4. Example PPS production rules. See text for explanation.

bidirectional flow of information: Control and specification information goes from the CPU to the input device, and the input device responds by transferring information from the external environment to the CPU's memory.

In a cognitive architecture, the input processing is similarly done by perceptual subsystems, but the control of input depends on some basic assumptions stemming from the apparent fact that only a fraction of the current environmental input should be responded to. That is, cognition must be protected from an overwhelming flood of external input, so the amount of information that cognition takes as input must be limited in some way. In EPIC, the input to cognition is the information in the perceptual store, so the question is whether and how the cognitive processor must deal with unneeded information in the perceptual store.

The traditional assumption in cognitive psychology is that a selection mechanism of some sort, usually named *selective attention*, controls what information is allowed to influence cognitive processing. In the box-model era, the mechanism was usually presented as some sort of filter between perception and cognition (see Norman, 1976). In current cognitive architectures, the attention mechanism typically uses activation levels to determine which condition items can trigger production rules. The idea of a selective attention mechanism has become so pervasive in cognitive theory that it seems odd not to simply adopt it, but it should be clear that the idea is not without its puzzles and problems, as consulting any textbook on the voluminous literature on classic topics such as early vs. late selection, and overt vs. covert attention should make clear. However, the actual phenomena of attention may be a result of a much wider range of mechanisms than an internal filter or activation. In particular, Findlay & Gilchrist (2003) point out how the concept of covert attention as been so heavily over-applied to visual perception that it has thoroughly obscured the more fundamental and powerful role played by the non-homogeneity of the retina, and voluntary eye movements that orient the high-resolution area of the retina on objects of interest. They argue that the role and effects traditionally assigned to covert visual attention are either subsumed by overt attention in the form of eye movements, or are very small compared to the effects of retinal non-homogeneity and eye movements. Thus a proper theory of attentional effects in vision would start with these fundamentals, and consider covert mechanisms only if there are important effects remaining.

As an alternative, EPIC assumes that when the characteristics of the peripheral systems, and how they can be cognitively controlled and accessed, are taken into account, the need for a separate information-limiting pre-cognitive selection mechanism disappears. In the case of vision, EPIC assumes that the perceptual store has a very large capacity, but the non-homogeneity of the retina "automatically" results in the visual system having the most information about the object being fixated, and substantially less about other objects. Thus the total amount of visual information is limited at the peripheral level, and the objects that have the most information are determined by the current eye location. The selective function is implemented not by some kind of special attentional mechanism, but simply by production rules that decide where more detail is needed, and command a corresponding eye movement, using relevant portions of the available visual information to guide the choice. Thus selection is handled simply in terms of what visual object the production rules single out for further processing, and what visual properties are used in this processing.

The first example rule in Figure 5, from a choice reaction task, illustrates this concept. The rule waits for a choice stimulus to appear, signaled by the detection of a visual *onset* event, which has been characterized as "capturing attention" (e.g. Yantis & Jonides, 1990). Visual onset events are assumed to be available over almost all of the visual field, so they will be detected even if the eyes are on some other object. This production rule adds a tag that the onset object is the choice stimulus, and then sets up the next production rules to fire. These will match on

```
(Waitfor_ChoiceStimulus
If ((Goal Do ChoiceTask) (Step Waitfor ChoiceStimulus)
   (Visual ?stimulus Detection Onset)
   (Not (Tag ?stimulus Choice_stimulus))))
Then ((Add (Tag ?stimulus Choice_stimulus))
   (Delete (Step Waitfor ChoiceStimulus)) (Add (Step Lookat ChoiceStimulus))))

(Top_Find_Hostile_blip
If ((Goal Monitor Situation) (Step Find Hostile_blip)
   (Visual ?blip Color Red) (Visual ?blip Shape Triangle)
   (Randomly_choose_one))
Then ((Add (Tag ?blip Current_track))))
```

Figure 5. Example of input selection in production rules. See text for explanation.

the tag for the Choice_stimulus to command the ocular processor to move the eyes to that stimulus object, and subsequent rules will use the tag to identify the object whose appearance will determine the response to be made. The second example rule in Figure 5 shows that the concept of selecting an object is not limited to events like onsets that "capture attention." This rule is from a radar-operator task model in which red triangular objects are "blips" that represent hostile aircraft, and so should be given priority. If the visual perceptual store contains an object that matches these perceptual properties, it is tagged as the current "track" for subsequent processing. In case there is more than one, one is chosen at random. In this case, other rules, following a different control structure, set up the next rules to fire, but these later rules will simply use the tag item to identify the selected object. Still other rules would deal with the possibility that a red triangular object might be present, but not visible given the current eye location.

The point of the examples is that there is no need for a distinct attention mechanism that pre-filters objects before cognition has access to them. Because the onset event, or the available perceptual properties, resulted in tagging a particular object as the stimulus, the subsequent response production rules can effectively ignore all other objects; the cognitive processor has been protected from excess and irrelevant information by a combination of peripheral limitations and a simple designation in working memory of which object is relevant for further processing.

Tagging a particular object as the one of interest could be described as a form of covert attention, and moving the eyes to an object could be called a case of overt attention. Regardless of these labeling issues, there is no actual "attention" mechanism in EPIC; there are only objects tagged for further processing, and information that is available or not, depending on the eye position. In line with previous warnings in the history of psychological theory, the concept of "attention" as used in current cognitive psychology may be a case of *reification*: just because there are a set of phenomena labeled "attention" does not mean that there is a distinct internal mechanism that "does" attention. Rather, attention could simply be our label for the processing performed by the human system when it responds selectively to events in the environment. EPIC shows how attention can be implemented without a dedicated internal mechanism that selects what information will be supplied to the cognitive processor.

Control of Output

In computers, output requires a bidirectional flow of information: the CPU must know when an output device is ready to accept commands before it issues them. Again in modern machines, dedicated hardware subprocessors handle all the detail work, but the basic concept is that of "handshaking" – the CPU checks for whether the output subsystem is free, sends the next output command if so, and then waits for the output subsystem to signal that the operation is complete before sending the next output command. This process became more advanced over the years, so that for example, the output subprocessor needs only to be told where the data is in memory, and upon command, it begins to autonomously output each item, and signals the CPU when it is finished outputting the entire set of data.

Similarly, in a cognitive architecture, the output is done by motor subsystems, but the cognitive processor must check the states of the motor processors and coordinate the usage of different output modalities to meet the task demands. In EPIC as originally proposed (Kieras & Meyer, 1997; Meyer & Kieras, 1997a), the motor processor for each modality required a relatively large amount of time to prepare each a movement by programming its movement features, and then the prepared movement could be separately initiated and executed[2]. Preparation of the next movement could commence while the previous movement was being executed, allowing EPIC to make a series of movements at high speed. However, a slow and deliberate style would wait for each movement to be completed before commanding the next movement. Thus EPIC's motor processors have two especially useful states: The *Modality Free* state means that the output modality (e.g. manual) is currently idle – a movement is neither being prepared nor is being executed. The *Processor Free* state means that any previously commanded movement has already started execution, so a new movement command will be accepted and prepared, and will execute as soon as the current movement (if any) is complete. Since the motor processor can prepare only one movement at a time, an error condition (*jamming*) will arise if it is commanded to prepare a new movement while it is still preparing one. By convention, a production rule that commands a movement always includes a condition that checks for whether the motor processor is in an appropriate state to prevent jamming.

Figure 6 and 7 illustrate the extremes in motor sequence programming. Note how by deleting and adding Step items, each rule disables itself and enables the next rule in the sequence. In the slow movement sequence shown in

[2] A reconsideration of the literature on motor feature programming currently underway suggests that aimed movements to visually identified objects do not require a substantial amount of time for motor feature preparation, but the situation for other kinds of movements is not yet clear.

Figure 6, each rule waits for the previously commanded movement to be complete before commanding the next movement, and the last rule waits for the last movement to be complete before proceeding. In the fast movement sequence shown in Figure 7, each movement is commanded as soon as the processor is willing to accept it, and the last rule does not wait for the movement to be complete. As one might suspect, skilled performance in a task appears to involve the fast movement regime, while the usual assumptions of simple forms of GOMS models (see below) appear to be consistent with a slow movement regime (see Kieras, Wood, & Meyer, 1997; Kieras & Meyer, 2000).

The point of these examples is that the cognitive strategy for how movements are marshaled and controlled can make a large difference in how rapidly a task can be performed, showing that some of the motoric aspects of a task are also a cognitive matter. Incorporating cognitive aspects of motor control is essential in modeling human performance, especially in highly interactive tasks such as computer operation.

Basic Control of Cognition

This section presents concept on the basic aspects of the flow of control in cognition, covering sequential, parallel, and interrupt-driven flow of control. The next section will consider complex control at the level of executive processes.

Sequential flow of cognition

In a computer, certain aspects of the sequential control of execution are primitive hardware functions. In particular, the machine instructions comprising a program are normally laid out in a series of consecutive memory locations. The hardware responsible for fetching and then executing each instruction assumes that the next

```
(Slow_wait_to_start
If ((Goal Make Responses) (Step make first)
    (Motor Manual Modality Free))
Then ((Send_to_motor Manual Perform Punch J Right Index)
    (Delete (Step make first))(Add (Step make second))))

(Slow_wait_for_first_done
If ((Goal Make Responses) (Step make second)
    (Motor Manual Modality Free))
Then ((Send_to_motor Manual Perform Punch K Right Middle)
    (Delete (Step make second))(Add (Step waiton second))))

(Slow_wait_for_second_done
If ((Goal Make Responses) (Step waiton second)
    (Motor Manual Modality Free))
Then ((Delete (Step waiton second))(Add (Step continue  working))))
```

Figure 6. Example of rules that produce a slow sequence of movements by waiting for each movement to be

```
(Fast_start_first
If ((Goal Make Responses) (Step make first)
    (Motor Manual Processor Free))
Then ((Send_to_motor Manual Perform Punch J Right Index)
    (Delete (Step make first))(Add (Step make second))))

(Fast_start_second_and_continue
If ((Goal Make Responses) (Step make second)
    (Motor Manual Processor Free))
Then ((Send_to_motor Manual Perform Punch K Right Middle)
    (Delete (Step make second))(Add (Step continue working))))
```

Figure 7. Example of rules that produce a fast sequence of movements by commanding each movement as soon as the motor processor is free.

instruction is normally in the next memory location; the hardware *program counter* or *instruction address register* is thus automatically incremented to point to the next instruction in memory. A program *branch* or *go to* instruction specifies an address for the next instruction that is not simply the next location; dedicated hardware will then set the program counter to that next address directly. However, there have been computers whose memory was organized differently so that sequential memory locations could not be assumed. An example is early machines that used a rotating magnetic drum as the main memory, where the delay waiting on the drum to rotate to any one position was substantial. Each instruction included the address of the next instruction, allowing them to be located on the drum in positions that minimized the delay in fetching the next instruction.

Production rule architectures are more like the early drum storage machines than modern machines, in that the results of the actions in each production rule normally control which rule will fire next. In EPIC, in fact, the flow of control must be fully explicit in the rules, because no flow of control is implicit in the PPS interpreter. That is, since any number of rules can fire on any cycle, the only way to control which rules will fire is to write the rule conditions and actions so that the contents of working memory cause the right rules to fire on the right occasions in the right sequence.

To bring some organization to PPS rules, Kieras & Polson (1985) and Bovair, Kieras, & Polson (1990) worked out how to organize production rules in terms of GOMS models (Card, Moran, & Newell, 1983; John & Kieras, 1996; Kieras, 1988, 1997, 2004), which have been extremely useful in representing human procedural knowledge in models of human-computer interaction. The basic idea can be explained by explaining the acronym: procedural knowledge is well-represented in terms of ***G****oals* that can be accomplished by executing a series of ***O****perators* organized in terms of ***M****ethods*, which are sequences of operators. ***S****election Rules* choose between alternate methods for the same goal. Kieras & Polson realized that it is useful to distinguish goals and operator sequence; that is, a method for accomplishing a goal specifies a series of operators, conveniently grouped into steps. Thus basic sequential flow of control for EPIC models is provided using the scheme shown in Figure 8 (and anticipated by some of the earlier examples). If working memory starts with the contents (Goal Top) and (Step T1), the first rule, named Top1, will fire, and its actions will delete (Step T1) and add (Step T2). At this point, Top1 can no longer fire, since one of its condition items has been deleted, but the rule Top2 can now fire. Similarly, it disables itself and enables the next rule, Top3, and so forth. The entire series of rules is under the control of the goal item in memory, (Goal Top); other rules can manipulate the Goal item to control this method. For example, suppose the first rule fires at the same time as the Goal item is removed by some other rule; the (Step T2) item will be present, but rules will be disabled and the next rule Top2 will not fire. If the Goal item is replaced, execution will resume where it left off; Top2 will fire, followed by Top3. EPIC's theory of executive processes is based on this concept of controlling methods by removing or replacing their controlling Goals.

The other basic form of flow of control is subroutine call/return, whose role in re-using and organizing program code is fundamental to computer programming methodology. On a computer, when a call to a subroutine is made, the address of the next instruction is saved, and a branch is made to the address of the first instruction of the subroutine. When the subroutine completes its work, it returns by branching to the saved address of the next instruction in the calling code. In addition to this basic flow of control scheme, arrangements are made to give the subroutine temporary memory space and pass input and output values between the calling routine and the subroutine. While subroutine call and return can be done with simple instructions, most computer hardware has specialized instructions to allow this very common process to be executed quickly and with only a few instructions.

```
(Top1
If ((Goal Top)(Step T1))
Then ((Delete (Step T1))(Add (Step T2))))

(Top2
If ((Goal Top)(Step T2))
Then ((Delete (Step T2))(Add (Step T3))))

(Top3
If ((Goal Top)(Step T3))
Then ((Delete (Step T3))(Add (Step T4))))
```

Figure 8. Schematic illustration of basic sequential flow of control.

Again in the GOMS model, accomplishing goals typically involves accomplishing sub-goals, and the resulting hierarchy of methods describe the natural units or modules of procedural knowledge. Thus GOMS models make heavy use of an analogue of subroutine call and return, and EPIC models are programmed correspondingly using a set of conventions illustrated in Figure 9. The basic scheme can be termed *hierarchical-sequential*. The methods are arranged in a strict calling hierarchy, and the execution is strictly sequential. Each rule that is part of a method or submethod includes a condition that names the goal for the method. Thus the rules for a method can only fire if the corresponding goal is being accomplished.

Rule Top1 calls a submethod by adding the goal for the submethod, (Goal Sub) in this example. It disables itself and enables the next rule in the calling sequence. This next rule includes a negated condition (Not (Goal Sub)), which means that the rule can fire only if the (Goal Sub) item is *not* present in working memory. Thus the next rule will wait until the subgoal item has been removed. By convention, the Step items for a submethod include the subgoal name, so when the (Goal Sub) item is placed in working memory by the Top1 rule, the rule Sub_startup fires because there are no step items belonging to this method. The startup rule enables the next rule in the method, which enables the next rule, as usual. Thanks to this scheme, the calling method does not need to know which Step item governs the first step of the submethod, which makes it easier to write re-usable methods that represent sharable pieces of procedural knowledge. Note how the submethod rules are all governed by the same (Goal Sub) item, so when the final submethod rule Sub2_terminate fires, the (Goal Sub) item and the last step are deleted; this indicates that the submethod has accomplished its goal, and the submethod rules are now all disabled. The disappearance of the subgoal triggers the next rule in the calling sequence, Top2, which then goes on to execute its next step.

Like most production-system cognitive architectures, EPIC assumes that working memory is shared across all of the production rules. Thus, the concepts in conventional programming languages of local variables, parameters, and return values do not apply – it is as if all the information is in global variables, accessible to all parts of the program. This presents some technical problems in writing re-usable sets of production rules, but it is not necessary to discuss them for the purposes of this chapter.

Parallel Flow

Parallelism is routine in computers. Perhaps the most distorted part of the computer metaphor is the notion that computers do only one thing at a time. While the capacity for parallel operation in computers has been steadily increasing, it is hardly a new capability. In fact, even the classic ENIAC was actually a parallel multiprocessor machine in which the separate computational modules were operating simultaneously.

```
(Top1_call_submethod
If ((Goal Top)(Step T1))
Then ((Add (Goal Sub))(Delete (Step T1))(Add (Step T2))))

(Top2_wait_for_return
If ((Goal Top)(Step T2)(Not(Goal Sub))
Then ((Delete (Step T2))(Add (Step T3))))
    ... ... ... ... ... ...

(Sub_startup
If ((Goal Sub) Not(Step ???))
Then ((Add (Step S1))))

(Sub1
If ((Goal Sub) (Step S1))
Then ((Delete (Step S1))(Add (Step S2))))

(Sub2_terminate
If ((Goal Sub) (Step S2))
Then ((Delete (Step S2))(Delete (Goal Sub))))
```

Figure 9. Schematic illustration of hierarchical-sequential flow of control used in subroutine calls.

Current computers are usually classified as being either *uniprocessor* machines, containing only a single CPU, or *multiprocessor* machines, with multiple CPUs. However, even uniprocessor computers have considerable parallelism. As mentioned above, computers have long had subprocessors for input/output that run in parallel with the CPU. The internals of the CPU typically have also involved parallel data movement, but in modern CPUs, the subparts of the CPU also operate in parallel. For example, in *pipelining* and *superscalar architectures*, the CPU is executing more than one instruction simultaneously, and not necessarily even in the original order!

Multiprocessor machines either have a shared main memory, or communicate data over a network, and display true parallel program execution with their multiple CPUs – they can literally execute two machine instructions simultaneously, and thus execute two programs truly simultaneously. Such machines are not new (commercial versions appeared in the 1960s), but they have become quite inexpensive, as shown by the common dual-CPU desktop computer.

Parallel flow of control in EPIC. Despite its routine presence in computers for decades, and the obviously massively parallel operation of the brain, parallel processing in cognitive theory has traditionally been treated as an exotic capability, with sequential processing assumed to be the normal mode of cognitive operation. There might be a variety of reasons for this bias, but most directly relevant to this chapter is the issue pointed out by Meyer and Kieras (1997a, b; 1999): There has been a long-standing tendency to mistake optional strategies adopted by subjects for "hardwired" architectural limitations that impose sequential processing. For example, in many dual-task paradigms, the task instructions and demands actually encourage unnecessarily sequential processing; in fact, different instructions and payoffs can encourage parallel processing (Meyer, Kieras, Lauber, Schumacher, Glass, Zurbriggen, Gmeindl, & Apfelblat, 1995; Meyer, Kieras, Schumacher, Fencsik, & Glass, 2001; Schumacher, Lauber, Glass, Zurbriggen, Gmeindl, Kieras, & Meyer, 1999; Schumacher, Seymour, Glass, Fencsik, Lauber, Kieras, & Meyer, 2001). Thus rather than a fundamental architectural feature, the presence of cognitive sequentiality is a result of a lack of strategic sophistication in the production rules in effect for a task. Against this background, it is useful to explore a different theoretical paradigm in which cognitive parallelism is *normal* rather than exotic and unusual.

EPIC directly supports full parallel processing since multiple production rules can fire on a cycle simultaneously and independently, and since Goals and Steps are simply working memory items, multiple ones can be present simultaneously – there is no *stack* that requires only one goal to be active at a time. Thus different *threads of execution* can be spawned and terminated at will. The previous Figure 9 illustrating hierarchical-sequential flow of control can be compared with Figure 10, which shows simultaneous execution of a top-level method and a submethod. The first rule in Figure 10 activates a submethod by adding its goal to working memory, but execution simultaneously continues with the next step in the calling method, until the rule for Step 8, which tests for the submethod having accomplished and removed its goal. Thus both the top-level goal and the subgoal are simultaneously pursued, each in its own thread of execution (see below for the corresponding *thread* concept in computing), but these are brought back into synchrony before Step T9 at the top level.

A few more schematic examples will illustrate the flexibility of parallel control in EPIC. Figure 11 shows how two subgoals could be simultaneously executed in parallel with the top level, which in this example waits for both of them to complete before proceeding; of course, this is optional – the submethods could simply be allowed to complete while other processes continue. Figure 12 shows single-step multiple threads within a method. The first

```
(Top1_start_submethod
If ((Goal Top)(Step T1))
Then ((Add (Goal Sub))
   (Delete (Step T1))(Add (Step T2))))

(Top2_keep_processing
If ((Goal Top)(Step T2))
Then ((Delete (Step T2))(Add (Step T3))))

   ... ... ... ... ... ...

(Top8_wait_for_submethod_complete
If ((Goal Top)(Step T8)
   (Not(Goal Sub))
Then ((Delete (Step T8))(Add (Step T9))))
```

Figure 10. Schematic illustration of overlapping hierarchical parallel flow of control.

```
(Top1_start_two_submethods
If ((Goal Top) (Step T1))
Then ((Add (Goal Sub1))(Add (Goal Sub2))
   (Delete (Step T1))(Add (Step T2))))

(Top2_wait_for_both_complete
If ((Goal Top)(Step T2)
   (Not(Goal Sub1)(Not(Goal Sub2)))
Then ((Delete (Step T2))(Add (Step T3))))
```

Figure 11. Schematic illustration of parallel submethods.

```
(Top1
If ((Goal Top)(Step  T1))
Then ((Delete (Step  T1))(Add (Step  T2))))

(Top2_branch_1
If ((Goal Top)(Step  T2) /* some condition */)
Then (/* some action */))

(Top2_branch_2
If ((Goal Top)(Step  T2) /* some condition */)
Then (/* some action */))

(Top2_continue
If ((Goal Top)(Step  T2))
Then ((Delete (Step  T2))(Add (Step T3))))
```

Figure 12. Schematic illustration of multiple threads within a single step.

rule enables all three different rules for the next step; one always fires to continue to the next step, but either, both, or neither of other two rules might fire as well. Figure 13 shows multiple threads within a method. The first rule spawns two threads in the form of different Step items, each of which starts a chain of rules governed by the same Goal item. If desired, these chains can simply be left to terminate on their own.

One often-cited advantage of a production rule representation for procedural knowledge is *modularity* – the independence of each production rule from other production rules. By allowing more than one rule to fire at once, it is possible to take more advantage of this modularity to provide a variety of flexible control structures.

```
(Top1_spawn_threads
If ((Goal Top)(Step  T1))
Then ((Delete (Step  T1))(Add (Step T2))
   (Add (Step ThrA1))(Add (Step ThrB1))))

(TopThreadA
If ((Goal Top)(Step ThrA1))
Then ((Delete (Step ThrA1))(Add (Step ThrA2))))

(TopThreadB
If ((Goal Top)(Step ThrB1))
Then ((Delete (Step ThrB1))(Add (Step ThrB2))))
```

Figure 13. Schematic illustration of multiple threads within a method.

Interrupt Control of Cognition

Computers have long had special *interrupt* hardware to allow a rapid response to external events that happen *asynchronously*, that is, at any time not necessarily in synchrony with activities of the computer. An interrupt signal will force a branch to a special address, where interrupt-handling code determines the cause of the interrupt, takes appropriate action, and then branches back to the original execution address. The hardware can be very elaborate, for example, automatically giving precedence to the highest-priority signal. Interrupts can be routine, or extraordinary. For example, control of input/output devices is routinely done with interrupts that signal that the device has completed an operation and is ready for the next datum or instructions. In contrast, some error conditions, such as an attempt to execute an invalid instruction, might cause an interrupt that results in a process being terminated, or even the OS entering an error state and waiting for a restart. Thus handling an interrupt spans a wide range of possible activities.

A production-system cognitive architecture can support interrupt handling, but only if the architecture has the appropriate properties. For example, it has to be possible for a rule to fire regardless of whatever else is going on, and the actions of the rule must be able to modify the state of the current processing, such as suspending other activities and starting new ones, essentially a form of executive process, which will be discussed later. Not all architectures can support interrupt rules in a straightforward way. EPIC's cognitive parallelism and representation of control items in working memory makes interrupt rules very simple.

The schematic first rule in Figure 14 illustrates a simple form of interrupt handling. This rule will fire unconditionally when a red object is present in the visual working memory, regardless of what other activity is underway. The second rule is governed by the goal of processing a blip in a radar-console task. If a visual object tagged as the current blip begins to disappear from the screen, the rule will delete the goal of processing the blip, and start a method that aborts the processing. A rule like this is needed in tasks where a visual object that is the subject of extended processing might disappear at any time; if the object disappears, the processing of it needs to be terminated and some "cleanup" work needs to be done. Such a rule is a simple way to handle what would otherwise be extremely awkward to implement in a strictly sequential architecture; it works because it can fire at any time its conditions are met, and its actions can control the state of other processing.

```
(Watchfor_red
If ((Visual ?obj Color Red))
Then (/* some action */))

(Watchfor_disappearing_blip
If ((Goal Process Blip)
   (Tag ?blip Current_blip)(Visual ?blip Status Disappearing))
Then ((Delete (Goal Process Blip))(Add (Goal Abort Processing))))
```

Figure 14. Sample interrupt rules.

Executive Control of Cognition

What is Executive Control?

In psychology, the term "executive" is vague and wide-ranging, having been applied in ways as suspect as a synonym for the homunculus, to being a specialized activity such as working memory management (see Monsell & Driver, 2000). In the sense used here, the executive is the process that controls what the human system as a whole is doing, such as what tasks are being performed, what resources are being used for them, and which tasks take priority over others, and how simultaneous tasks should be coordinated. In computers, this function is most similar to Operating Systems, so it is worthwhile to present a quick survey of the key concepts. See Tucker (2004) or any modern Operating Systems textbook, such as Stallings (1998), for more detail.

Survey of Operating System Concepts

Operating Systems (OSs) originated at a time when computer hardware, especially the CPU, was vastly more expensive than it is now, so there were huge economic incentives to wring every last bit of performance from the investment by clever programming. Of course today the hardware is cheaper, but the demands are higher, so OS technology has continued to develop. The key trick is that since the CPU is tremendously faster than peripheral

input/output devices, it is possible for the CPU to be executing instructions for one task while waiting for an input/output operation for a different task to complete. Thus two data processing tasks could progress simultaneously by overlapping CPU execution of one with input/output waits for the other.

This overlapping could be accomplished by custom-building a single complex program for each possible combination of computing jobs, but for almost all purposes, such detailed, difficult, and tedious programming work would be prohibitively expense. After various partial solutions (e.g. spooling) a grand generalization emerged around 1970, termed *multiprogramming*, or *multiprocessing*, that made it possible to write software for simultaneous or *concurrent*, task execution quite easily.

The basic idea is that a control program would always be present in memory, and would keep track of which other programs were present in memory, what their states were, and which resources were currently associated with them. This control program was at first called a *resident monitor*, later the OS. With the help of interrupts, the OS code can switch the CPU from one program to another by controlling where execution resumes after handling an interrupt.

The OS manages programs in terms of *processes*, which are a set of resources, code in memory, data in memory, and an execution state or pathway. The resources include not only peripheral devices such as disk drives, but also space in memory and CPU time. When a program needs access to a resource, it makes a request to the OS, and waits until the request is granted. Because it is thus informed of resource needs and has all the required information available about all the processes, the OS can manage multiple processes in such a way that they can be simultaneously active as long as their resource requirements do not conflict, and that when conflicts do occur, they are resolved according to the specified priorities.

Two processes are *non-cooperating* if they do not attempt to communicate with each other or share resources. In this case, as long as the protocol for accessing resources through the OS is followed, the program for each process can be written independently, as if it was going to be the only program executed; the OS will automatically interleave its resource usage with that of other independent processes, and will ensure that they will not interfere with each other, and that the maximum throughput consistent with process priorities will be obtained. This is the major contribution of OS technology – enabling concurrent operation with simple programming techniques. Anybody who has written and run even simple programs on a modern computer has in fact used these facilities.

Cooperating processes are used in specialized applications and communicate with each other or access a shared resource such as a database. For such situations, the OS provides additional services to allow processes to communicate reliably with each other, and to ensure mutual exclusion on critical resources – such preventing two processes from attempting to modify the same area of memory simultaneously. Using these interprocess communication and mutual exclusion facilities properly in specialized applications is the hard part of multiprogramming technology.

One last distinction concerns the relationship between execution pathways and resources. Switching the CPU between processes can be very time-consuming since all of their resources might have to be switched as well. For example, if a low priority process gets suspended in order to free up required memory space for a high priority process, the OS will need to copy the program code, data, and execution state information for the low priority process out to disk storage, and then read it back into memory when it is time to resume the process (virtual memory hardware helps automate this process). In contrast, a *thread* is a *lightweight process* – it is an execution pathway that shares resources with other threads in the same process. The OS can switch execution between threads very quickly because some critical resources such as memory space are shared. Multithreaded programming is becoming steadily more common as a way to implement high-performance applications, but because the threads are usually cooperating, OS facilities for inter-thread communication and mutual exclusion are usually involved.

A key insight from OS methodology is that the key algorithms work for both uniprocessor and multiprocessor machines. While not explicitly acknowledged in OS textbooks, multiprogramming on a uniprocessor machine is essentially simulated parallel processing, so OS operations are similar for both true versus simulated parallel processing. Closely related is that fact that all of an OS's capabilities can be implemented with very little specialized hardware support, although such hardware can improve efficiency and simplify the programming.

How should Executive Processes be Represented in a Cognitive Architecture?

Traditionally, executive processes are represented in a cognitive architecture as specialized built-in mechanisms. A well-known example is the *Supervisory Attentional System* in the Norman and Shallice (1986) proposal, which is a specialized brain system whose job it is to pick the most activated schema and cause it to take control of cognition. A more contemporary and computational example is Salvucci's (Salvucci, Kushleyeva, & Lee, 2004) proposals to

add OS-like facilities into the core functionality of the ACT-R architecture. Proposals of this sort, which propose special-purpose "hardware" support in the architecture for executive processes, overlook the lesson from computing that minimal hardware support is necessary for OSs to work well. More seriously, they also overlook a wealth of literature on how executive control is a strategic process that is a result of learning and can be modified on the fly by changes in task priorities or instructions (for reviews, see Meyer & Kieras, 1997a,b, 1999; Kieras, Meyer, Ballas, & Lauber, 2000), which suggests that executive processes are similar to other forms of cognitive skill, and so should be represented simply as additional procedural knowledge rather than specialized architectural components.

EPIC's solution is that simply by making task goals another kind of working memory item, and allowing more than one production rule to fire at a time, it is easy to write production rules that control other sets of production rules by modifying which goals are currently present in memory. Thus, no special-purpose architectural mechanisms are required for executive control. The parallelism is critical because it ensures that the executive rules and the individual sets of task rules can all execute simultaneously, meaning that the executive process rules can intervene whenever they need to, and the individual task rules do not need to constantly check for the states of other processes to allow for changes in which processes have control.

Thus the cognitive modeler can work like the modern computer programmer in that as long as the production rules for non-cooperating processes observe a protocol of asking the executive process for access to resources, they can be simply written as if they were the only production rules being executed at the time. Again analogously to modern computer programming, this allows sets of production rules for different tasks to be written as modules that implement principles of abstraction, encapsulation, and re-usable code; the fact that production rules can thus be organized into independent modular sets and flexibly combined has implications not just for the practicalities of modeling, but also for how humans might learn multitasking skills (see Kieras, Meyer, Ballas, & Lauber, 2000).

Executive Processes in EPIC

In EPIC an executive process is simply a set of production rules that control other sets of production rules by manipulating goal information in working memory, along with possibly other information as needed to control the execution of other production rules. This section discusses some distinctions relevant to the representation of executive processes.

Specialized vs. general executives. Kieras et al. (2000) made a distinctive between *specialized executive* processes that contain task-specific rules for coordinating two or more specific tasks. For example, the executive rules could pre-position the eyes to the locations where input for the tasks will appear, improving performance. A specialized executive could monitor the state of one task and modulate the other task to improve performance by changing specific control items in memory. The executive processes describing in the Meyer & Kieras (1997a, b) models for the psychological refractory period used this mechanism to account for the fine quantitative details in a high-precision data set. Making use of task-specific information means that the executive process production rules can be tuned to permit the maximum possible performance, but the details of the rules would only apply to that specific pair of tasks and their exact conditions. In addition, the production rules for each task will have to be written to take into account how they will be controlled by the executive process, typically in very specific ways.

In contrast, a *general executive* process can coordinate any set of subtasks because the executive process rules do not have to be changed to suit particular subtasks. The production rules for each task will have to follow the general executive's protocol in order to allow them to be controlled by the general executive, but do not otherwise have to take into account which other tasks might be running at the same time. Usually, a general executive will be less efficient than a specialized executive, because it will not take advantage of any special characteristics of the situation to enable higher performance, but on the other hand, the general executive production rules are fully general, and do not have to be acquired and tuned for each new combination of tasks.

Sequential vs. concurrent executives. The overall pattern of executive control of two or more tasks can be ranged from most sequential to most concurrent. A *single-task executive* ensures good performance on the highest priority task simply by ignoring the lower-priority tasks; the result is best-possible performance on the most important task. A *sequential executive* runs one task, then the other, and then repeats. This is a simple scheme for dealing with resource conflicts – only one task runs at a time, so conflicts are not possible. However, both of these simple solutions will underutilize the system resources since they will take no advantage of the possibilities of interleaving tasks.

An interesting case is a *time-slicing executive*: on each cycle, an extremely simple executive removes the goal for one task, and inserts the other, so that each task is enabled on alternate cycles. To prevent resource conflicts, the production rules for each task must follow the protocol of not attempting to use a resource (such as a motor processor) that is already busy, but do not otherwise have to interact with the executive process or each other. Each

task is thus allowed to proceed as long as it uses resources that are free. This is a purely sequential executive that interleaves resource utilization between the tasks on an as-needed first-come/first-served basis. This scheme requires that each task can be suspended at any point without conflicts or loss of information, and the concurrent tasks be of equal priority. In some sample dual-task models, this scheme works surprising well because most of the time for the tasks consists of waiting on motor processors to complete their activity; the one-cycle delay in executing the production rules for each task impairs individual-task performance only very slightly, while the overall dual-task performance benefits by the automatic and very efficient interleaving of motor processor usage.

A fully *parallel executive* runs both tasks simultaneously, and intervenes only when necessary to prevent resource conflicts and enforce task priorities. Such an executive requires that each task interact with the executive before attempting to use a resource; only in this way can the executive allocate resources based on the relative priorities of the tasks.

Executive protocols. To run in an environment in which an executive process is in control, the task rules have to be written according to a protocol of some sort. Under EPIC, this requires that the tasks rules be governed by a single goal that an executive can manipulate to suspend or resume the task as a whole. In addition, some protocol governing access to resources must be followed. Under EPIC, a standard protocol is that a motor processor must not be commanded unless it is in a free state, as described previously. These basic assumptions lead to three kinds of executive protocols, described in terms of what kind of executive can successfully manage the task rules.

Single-task compatibility means that the task rules are written as if all resources were available for use by this set of rules at any time. The single-task, sequential, and time-slicing executives can successfully coordinate tasks that follow only this level of compatibility, but parallel executives cannot.

Specialized-executive compatibility means that the task rules assume the presence of a specific specialized executive that for example, places specific control items in memory or do some of the task-specific processing. Such task rule sets are completely tied to a specialized executive, which in turn is bound to a specific combination of task rules.

General-executive compatibility means that the task rules follow a general executive protocol for resource access; before attempting to use a resource, they always request allocation of that resource from the general executive, wait for the resource to be allocated, and then release the resource as soon as possible, or accept pre-emption. If the task rule sets follow this protocol, then a general executive can reliably coordinate resource usage for any pair of tasks.

Examples of Executive Processes

This section provides concrete examples of the executive concepts and distinctions just presented. A model for a dual task situation will be presented in each of two executive control regimes: a specialized executive, and a general executive. First the task will be described, then the models presented.

The "Wickens" dual-task experiment. The Martin-Emerson & Wickens (1992) task was first used for EPIC modeling as reported in Kieras & Meyer (1995), and since then has been re-used by other modelers. The task is a simple dual-task, involving a two-choice reaction task and a basic compensatory tracking task. Kieras & Meyer originally chose it for modeling because it involved a manipulation of the visual relationship between the two tasks, with eye movements sometimes being required.

Figure 15 shows the view of the display in the EPIC simulation package. The small square at the top of the display is the tracking target; the small cross is the tracking cursor. The subject's task is to keep the cross on the target by moving a joystick, despite a forcing function that tries to randomly displace the cursor. The dark circle imposed on the target represents the one-degree foveal radius, showing that the eye is currently fixated on the target. The large concentric circle is a 10-degree calibration circle. The one-degree circle near the middle of the display marks where the choice stimulus will appear, which it does at random intervals.

In Figure 16, the choice stimulus has appeared; it is either a right- or left-pointing arrow, and the subject is required to press the corresponding one of two buttons under fingers on the left hand. However, whether the arrow orientation can be discriminated visually depends on the distance between it and the current eye position; Martin-Emerson and Wickens manipulated this over a wide range. For reasonable retinal availability characteristics, the eye must be moved for eccentricities like those illustrated. Figure 17 shows the position of eye on the choice target. Once the choice stimulus is identified, the response button and be pressed, and the tracking task resumed, returning to the situation shown in Figure 15. Dependent variables were the choice response latencies and tracking error accumulated during a two-second period starting when the choice stimulus appeared.

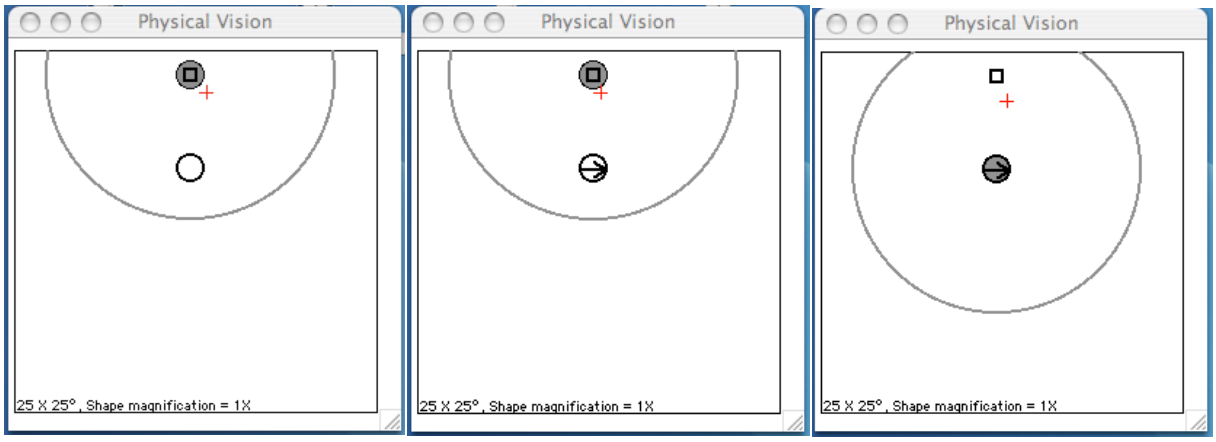

Figure 15 (left). The "Wickens" task during tracking, as shown in EPIC's display of the physical visual field. The small gray circle represents the position of the fovea; the larger concentric circle is a 10-degree calibration circle.

Figure 16 (center). The choice stimulus has appeared in the marker circle.

Figure 17 (right). The eye has been moved to the choice stimulus.

Resource allocation in the Wickens task. Both tasks require use of the ocular motor processor for acquiring the stimulus (which implies use of fovea for each stimulus), and the manual motor processor for making the response. The motor processors are thus the critical resources, since under EPIC's assumptions, a motor processor cannot simultaneously perform two different actions. Thus the two tasks must share them without conflict, and the efficiency of this sharing determines the overall dual task performance. If each task is implemented with its own set of production rules, then there must be an executive process that coordinates the execution of each set of task rules and their access to the ocular and manual motor processors.

Before going further, it is worth discussing an alternative to an executive process: Suppose the human simply learns a single rule set for the Wickens task as a single complex combined task. That is, instead of trying to concurrently perform two distinct tasks, corresponding to two goals (Goal Do Tracking) and (Goal Do Choice), the human has only a single goal and its corresponding production rules: (Goal Do Combined_Task), and these production rules emit the appropriate sequence of motor movements for joystick manipulations, button pushing, and eye movements as required to respond to cursor movements and arrow appearances. As Kieras et al. (2000) argue, such a combined production rule set might well be a final end state of learning after considerable practice, but the dual task representation seems more likely to be the state of learning in typical dual-task experiments where only a moderate amount of practice has been done. At a purely theoretical level, if there are not separate production rules for each task, it means that we do not have a dual-task situation, but rather a complex single task that does not require the services of an executive. Thus, at least for the sake of theoretical exploration of executive processes, we need to continue with the assumption of separate task rule sets for each task.

Model components. To construct an EPIC model for the Wickens dual-task, we need four components: (1) A set of Startup rules that run at the beginning of the experiment to identify and tag the different screen objects for use in the other rule sets. (2) A set of rules for tracking, governed by (Goal Do Tracking). In this example, these can be written in two threads: one keeps the eye on the tracking target, while another moves the joystick whenever the cursor is too far from the target. Since this task is inherently continuous, these rules should be normally enabled by the more-or-less continuous presence of their governing goal in working memory. (3) A set of rules for the choice reaction task, governed by (Goal Do Choice). When the choice stimulus appears, the eye must be moved to the choice stimulus location, and then these rules select and execute the response depending on the shape property of the stimulus. This task is intermittent, but according to the experiment instructions, it is higher priority than the tracking task. (4) A set of executive rules for task coordination. In general form, they contain no information about the specific tasks, but in specialized form, they will need to use the choice stimulus onset as a trigger to determine when the choice task needs to run. For brevity, the following will focus on the executive rules, and show rules for the two sub-tasks only where important to explain the executive process.

Sequential specialized executive. Figure 18 shows an example of how a sequential specialized executive can coordinate the two tasks. For clarity of exposition, key items in the rules are emphasized with bold face. The first rule, Top_startup, invokes the Identify Objects submethod, whose rules are not shown for brevity. This adds tags to memory identifying the visual objects corresponding to the tracking target and the cursor,which the tracking task rules refer to. The next rule, Top_start_tasks adds the tracking task goal, which "turns on" the tracking task rules, and then enables the next rule, Top_waitfor_choicestimulus. This rule fires when there is a visual onset event, and tags the visual object as the choice stimulus, and suspends the tracking task by removing its goal. The next rule in sequence starts the choice task. When the choice task is complete, the final executive rule, Top_resume_tracking, replaces the tracking task goal and returns to waiting for the choice stimulus.

Figures 19 and 20 show the tracking and choice task rules written assuming that a specialized executive is present, but as if no other task was present. The tracking task rules are very simple, but actually run as two separate one-rule threads. The first, Watch_target, simply moves the eye to the tracking target whenever the target is outside the fovea. The second requires that the target be in the fovea, and moves the joystick with a Ply movement to place the cursor on the target whenever the distance between the target and the cursor (supplied by the perceptual processor) is greater than a specified amount. Note that the two rules can fire independently and repeatedly.

The choice task rules are more sequential in nature, being written for presentation clarity in the slow response style described previously. The first is a standard startup rule. The second moves the eye to the choice stimulus – which the specialized executive had detected and tagged as part of its activity. Again for simplicity, these rules always move the eye to the choice stimulus, even when it might be close enough to the tracking target to be discriminated in parafoveal vision. For brevity, of the next rules to fire, only one of the response selection rules is shown, the one that maps a left-pointing arrow to the corresponding response button-press. The final rule waits for the keypress response to be complete, and then terminates the method. Note again how these rules do not depend on any content of the other task of tracking, but do depend on the executive process to determine when these rules should run, and identify their stimulus in the process.

```
(Top_startup
If ((Goal Do Dual_task)(Not (Step ??? ???)))
Then ((Add (Goal Identify Objects))(Add (Step Start Tasks))))

(Top_start_tasks
If ((Goal Do Dual_task)(Step Start Tasks)
   (Not (Goal Identify Objects)))
Then ((Add (Goal Do Tracking))
   (Delete (Step Start Tasks))(Add (Step Waitfor ChoiceStimulus))))

(Top_waitfor_choicestimulus
If ((Goal Do Dual_task)(Step Waitfor ChoiceStimulus)
   (Visual ?stimulus Detection Onset))
Then ((Delete (Goal Do Tracking))
   (Add (Tag ?stimulus Choice_stimulus))
   (Delete (Step Waitfor ChoiceStimulus))(Add (Step Start ChoiceTask))))

(Top_start_choice_task
If ((Goal Do Dual_task)(Step Start ChoiceTask))
Then ((Add (Goal Do ChoiceTask))
   (Delete (Step Start ChoiceTask))(Add (Step Resume Tracking))))

(Top_resume_tracking
If ((Goal Do Dual_task)(Step Resume Tracking)
   (Not (Goal Do ChoiceTask)))
Then ((Add (Goal Do Tracking))
   (Delete (Step Resume Tracking))(Add (Step Waitfor ChoiceStimulus))))
```

Figure 18. Specialized sequential executive rules for the Wickens task.

```
(Watch_target
If ((Goal Do Tracking)(Step Watch Target)
   (Tag ?target Target)(Not (Visual ?target Eccentricity Fovea))
   (Motor Ocular Modality Free))
Then ((Send_to_motor Ocular Perform Move ?target)))

(Ply_cursor
If ((Goal Do Tracking)(Step Track Target)
   (Tag ?cursor Cursor)(Tag ?target Target)
   (Visual ?target Eccentricity Fovea)
   (Visual ?cursor Distance ?error)(Greater_than ?error .1)
   (Motor Manual Modality Free))
Then ((Send_to_motor Manual Perform Ply ?cursor ?target Right)))
```

Figure 19. Tracking task rules that assume a specialized sequential executive.

```
(Choice_task_startup
If ((Goal Do ChoiceTask)(Not (Step ??? ???)))
Then ((Add (Step Lookat ChoiceStimulus))))

(Lookat_choice_stimulus
If ((Goal Do ChoiceTask)(Step Lookat ChoiceStimulus)
   (Tag ?stimulus Choice_stimulus)
   (Motor Ocular Modality Free))
Then ((Send_to_motor Ocular Perform Move ?stimulus)
   (Delete (Step Lookat ChoiceStimulus))(Add (Step Waitfor EyeDone))))

. . .

(Select_response_left
If ((Goal Do ChoiceTask)(Step Select Response)
   (Tag ?stimulus Choice_stimulus)
   (Visual ?stimulus Shape Left_Arrow)
   (Motor Manual Modality Free))
Then ((Send_to_motor Manual Perform Punch B1 Left Middle)
   (Delete (Step Select Response))(Add (Step Finish Task))))

. . .

(Finish_choice_task
If ((Goal Do ChoiceTask)(Step Finish Task)
   (Tag ?stimulus Choice_stimulus)
   (Motor Manual Modality Free))
Then ((Delete (Tag ?stimulus Choice_stimulus))
   (Delete (Step Finish Task))
   (Delete (Goal Do ChoiceTask))))
```

Figure 20. Choice task rules that assume a specialized sequential executive.

The sequential specialized executive thus solves the resource allocation problem by forcing the tasks to execute sequentially – the higher priority choice task runs to completion while the tracking task is suspended. Each subtask rule set can then follow a simple protocol for accessing the motor processors, relying on the specialized executive to ensure that they are not in conflict. The disadvantage is that the tracking task is suspended for the entire time required for the choice task, which in the example presented here, is substantial. This executive is necessarily specialized because it must "know" something about the choice task – namely, what stimulus is the trigger for

starting execution of the choice task. There is little additional harm done by having the specialized executive provide the stimulus specification to the choice task.

Parallel general executive. This form of executive is most like a modern computer OS in that it performs general resource management during parallel task execution. Both of the tasks are running constantly – both goals are always present in memory. One task goal is marked as higher priority than the other. Each task asks for permission to use a motor processor, whereupon the executive grants the request if the resource is free, or withholds permission if not. Each task releases a resource as soon as it is finished with it, and the executive reallocates the released resource to a task that is waiting for it. In case of conflict, the executive causes a higher-priority task to preempt a lower-priority task. The advantage to this general executive is that the task rules are independent of each other, and do not rely on the executive to handle any of the task specifics as in the previous example. The disadvantage is that the ask/wait/release processing each task must perform prior to using a resource constitutes some overhead. However, at least in the case of two tasks, this overhead can be minimized with a broad-but-shallow set of executive rules.

Logically, it is impossible to do away with specialized executive knowledge completely, in that there must always be some executive rules that specify which specific tasks should be performed in a given situation and what their priority relationship is. These are shown in Figure 21. As before, the first step (not shown here) is identifying the visual objects. The next rule Top_start_tasks then adds the goals for both the tracking task and the choice task, and marks the tracking task as being lower priority than the choice task. At this point, the general executive and the two sets of task rules can handle all of the task coordination, so no further top-level rules are shown.

Figure 22 shows the rules for the tracking task. A startup rule requests use of the ocular and manual motor processors by adding a *Status* item to memory. The Watch_target rule is the same as before, except its condition includes a Status item requiring that the tracking task has been allocated the ocular processor. Likewise, the Ply_cursor rule checks for permission to use the manual processor. Since the tracking task runs continuously but at a low priority, it never releases either processor – it is content to be pre-empted as necessary.

Figure 23 shows an excerpt of the rules for the choice task. The Waitfor_ChoiceStimulus rule is triggered by the stimulus onset, and requests use of the ocular processor. The executive process will note that the request is coming from the higher priority process, and will deallocate the ocular processor from the tracking task (which disables the Watch_target rule), and assign it to the choice task. The rule Lookat_choice_stimulus waits until permission is granted, then moves the eye to the choice stimulus, and releases the ocular processor. At this point the executive process reallocates the ocular processor back to the tracking task, re-enabling the Watch_target rule. The choice task rule Waitfor_shape fires when the shape of the choice stimulus becomes available, and requests the manual processor. In a similar manner, the executive withdraws the use of the manual processor from the tracking task and assigns it to the choice task, which then releases it in the same rule that commands the manual response.

To complete the example, Figure 24 shows a fragment of the general executive for coordinating any two tasks. The executive consists of many rules like those illustrated, one of which fires for each possible request or release situation, giving single-cycle response by the executive for resource requests and releases. For example, the first rule fires when the high priority task requests a resource currently in use by the low priority task, and its actions reassign the resource to the high-priority task, and note that the low priority task use of that resource has been suspended. The second rule fires when a task releases a resource when another task using that resource has been suspended, and reallocates the resource back to the suspended task.

Thus the use of the ocular and manual processors by the two sets of task rules are "automatically" interleaved by the general executive rules on an as-needed basis, and the resulting performance can be very high, even though there

```
(Top_start_tasks
If ((Goal Do Dual_task)(Step Start Tasks)
    (Not (Goal Identify Objects)))
Then (
    (Add (Goal Do Tracking))
    (Add (Status Trackingtask Priority Low))
    (Add (Goal Do ChoiceTask))
    (Add (Status Choicetask Priority High))
    (Delete (Step Start Tasks))
    (Add (Step Continue Running))))
```

Figure 21. Specialized task startup rule using a general executive.

```
(Tracking_task_startup
If ((Goal Do Tracking)(Not (Step ??? ???)))
Then (
   (Add (Status Trackingtask Request Ocular))
   (Add (Status Trackingtask Request Manual))
   (Add (Step Watch Target))(Add (Step Track Target))))

(Watch_target
If ((Goal Do Tracking)(Step Watch Target)(Tag ?target Target)
   (Not (Visual ?target Eccentricity Fovea))
   (Status Trackingtask Has Ocular)
   (Motor Ocular Modality Free))
Then ((Send_to_motor Ocular Perform Move ?target)))

(Ply_cursor
If ((Goal Do Tracking)(Step Track Target)
   (Tag ?cursor Cursor)(Tag ?target Target)(Visual ?target Eccentricity Fovea)
   (Visual ?cursor Distance ?error)(Greater_than ?error .1)
   (Status Trackingtask Has Manual)
   (Motor Manual Modality Free))
Then ((Send_to_motor Manual Perform Ply ?cursor ?target Right)))
```

Figure 22. Tracking task rules using a general executive.

```
. . .
(Waitfor_ChoiceStimulus
If ((Goal Do ChoiceTask)(Step Waitfor ChoiceStimulus)
   (Visual ?stimulus Detection Onset)(Not (Tag ?stimulus Choice_stimulus)))
Then ((Add (Tag ?stimulus Choice_stimulus))(Add (Status Choicetask Request Ocular))
   (Delete (Step Waitfor ChoiceStimulus))
   (Add (Step Lookat ChoiceStimulus))(Add (Step Waitfor Shape))))

(Lookat_choice_stimulus
If ((Goal Do ChoiceTask)(Step Lookat ChoiceStimulus)(Tag ?stimulus Choice_stimulus)
   (Status Choicetask Has Ocular)(Motor Ocular Processor Free))
Then ((Send_to_motor Ocular Perform Move ?stimulus)
   (Add (Status Choicetask Release Ocular))(Delete (Step Lookat ChoiceStimulus))))

(Waitfor_shape
If ((Goal Do ChoiceTask)(Step Waitfor Shape)
   (Tag ?stimulus Choice_stimulus)(Visual ?stimulus Shape ???))
Then ((Add (Status Choicetask Request Manual))
   (Delete (Step Waitfor Shape))(Add (Step Select Response))))

(Select_response_left
If ((Goal Do ChoiceTask)(Step Select Response)
   (Tag ?stimulus Choice_stimulus)(Visual ?stimulus Shape Left_Arrow)
   (Status Choicetask Has Manual)(Motor Manual Processor Free))
Then ((Send_to_motor Manual Perform Punch B1 Left Middle)
   (Add (Status Choicetask Release Manual))
   (Delete (Step Select Response))(Add (Step Finish Task))))
. . .
(Finish_choice_task . . .
Then ((Add (Step Waitfor ChoiceStimulus))))
```

Figure 23. Excerpt of choice task rules using a general executive.

is some overhead imposed by the general executive. This overhead can be minimal in the case of two tasks because the rules to adjudicate resource conflicts between two tasks are relatively simple and limited in number, and so can be written in the illustrated manner to involve no more than a single cycle of overhead. To implement the same single-cycle process for three or more tasks leads to a combinatorial explosion in rules and rule complexity, and the multiple-cycle equivalent would be very complicated and slow-executing.

Following up on the discussion of multitasking skill learning in Kieras et al. (2000), the complexity of all but the simplest general executive suggests that people have difficulty in multiple-task performance of novel task combinations primarily because they have not had occasion to tackle the difficult task of learning an efficient high-capacity general executive, leaving them to flounder in a complex multiple-task situation. Perhaps true general multitask executives are so difficult to learn that they rarely appear except in very simple situations, meaning that complex multiple tasks might always require learning a specialized executive before good performance can be produced.

Which Executive Process Model Should be Used?

The above sections present several basic aspects of executive processes and illustrate in detail how the different models of executive processes can be implemented as production rules. One obvious question is the choice of executive process representation: Which executive process should be used in a model? As warned at the beginning, this is not an empirical chapter, and this question is basically an empirical question.

But there are several reasons why it may be difficult or impractical to identify exactly what executive process is at work in a particular multiple-task data set. As alluded to previously, Kieras et al. (2000) point out how some executive strategies seem to be compatible with different levels of skill development, especially at the extremes. However, the executive process that should make it easiest to combine novel tasks would be the general parallel executive, but which in turn seems to require considerable learning. Thus the relationship of executive process type to state of practice may not be straightforward. In addition, it should be kept in mind that in a parallel cognitive architecture, the overt performance of the system will be more determined by degree of optimization of the perceptual-motor processes, and only slightly or not at all by the details of the control regime that implements that optimization. In other words, the differences between of the executive processes may not be on the critical path for determining task execution time. For example, the general parallel executive in the above example imposes only an occasional one-cycle overhead, and depending on the exact timings of the perceptual and motor activities, this overhead might not be visible in the manifested performance at all, and perhaps only in an extreme stable and

```
(GenEx_HP_Request_LP_Has_Preempt
If
((Status ?high_task Request ?resource)(Status ?high_task Priority High)
(Not (Status ?high_task Has ?resource))(Not (Status ?high_task Suspended ?resource))
(Not (Status ?high_task Waiting ?resource))(Not (Status ?high_task Release ?resource))
(Status ?low_task Has ?resource)(Status ?low_task Priority Low)
(Not (Status ?low_task Request ?resource))(Not (Status ?low_task Release ?resource)))
Then
((Add (Status ?high_task Has ?resource))
(Delete (Status ?high_task Request ?resource))
(Delete (Status ?low_task Has ?resource))
(Add (Status ?low_task Suspended ?resource))))

(GenEx_Reallocate_Resource_to_Suspended
If
((Status ?user Release ?resource)(Status ?user Has ?resource)
(Status ?low_task Suspended ?resource)
(Not (Status ??? Request ?resource))(Not (Status ??? Waiting ?resource)))
Then
((Delete (Status ?user Has ?resource))
(Delete (Status ?user Release ?resource))
(Add (Status ?low_task Has ?resource))
(Delete (Status ?low_task Suspended ?resource))))
```

Figure 24. Excerpt from the rule set for a parallel general executive.

precise data set. Finally, as argued above, much of the details in task performance are influenced by the subjects' strategies adopted under the influence of the task demands, which can be hard to control, or not controlled at all in many experimental paradigms.

Thus the different concepts of executive control are not offered necessarily as a better way to account for data, but rather as a better way to construct models. For example, the general parallel executive makes it very simple to construct a multi-task model that uses resources efficiently, and as argued in this chapter, there is no good reason to limit the cognitive processor in ways that would prevent its use. If it is difficult to demonstrate that any other executive model fits the data better, why not use the one that is the most theoretically productive and easy to use?

Another application of the executive process models would be in applying the *bracketing heuristic* proposed by Kieras & Meyer (2000) to models of multiple-task performance. This heuristic addresses the problem of how to predict performance in a task that depends heavily on individual task strategies, especially under conditions where no behavioral data is available to fit a model to, as in predicting human performance for purposes of system design. Alternatively, the heuristic provides a relatively easy way to explain effects in observed data where a well-fitting model would be hard to construct. The basic idea of the bracketing heuristic is to construct two a-priori models of the task: one is the *fastest-possible* model that uses a task strategy that produces the fastest possible performance allowed by the architecture. The other is the *slowest-reasonable* model uses a straightforward task strategy with no performance-optimizing "tricks". The predicted performance from these two models should bracket the actual performance. In a design analysis context, these predictions can be used to determine whether the system would perform acceptably despite the uncertainty about what level of performance the actual users will produce. The alternative use of the bracketing models is as an aid to interpreting data. Since the two bracketing models are well-understood, the data can be compared to them to arrive at a useful qualitative understanding of the effects in the data without the difficulties of iterating a single model to fit the data quantitatively, as illustrated in Kieras, Meyer, and Ballas (2001) who analyzed the tradeoffs manifested in two different interfaces for the same task.

The different executive process models can help specify the bracketing models: the fastest-possible model could either be some form of specialized parallel executive, but using a general parallel executive would probably produce very similar performance and be much easier to construct. The slowest-reasonable model could be represented as a simple sequential executive, reflecting a deliberate "do one thing at a time" approach. This approach would make the bracketing heuristic easier and more uniform to apply to the theoretical analysis of multiple-task performance.

Conclusion

Application of the modern computer metaphor to basic functions of cognitive control both clarifies some of the theoretical issues and opens up new areas for theoretical exploration. For example, the relation between the perceptual system and the cognitive processor can be characterized rather differently than the traditional attentional selection approach. Also, the heavy cognitive involvement in motor activity could be much further developed. But the main contribution of the modern computer metaphor is to open up the possibities for how cognition itself is organized and controlled, both in terms of basic flow-of-control issues such as parallelism and interrupts, and complex executive control of activity and resource allocation.

For many years, cognitive theory has discussed cognitive control, executive processes, and resource allocation as fundamental concepts in human multiple-task performance. Now we can directly model them, so that theories about these topics can develop more concretely, rigorously, and quantifiably. EPIC's flexible flow-of-control mechanisms makes these concepts easy to implement, meaning that we can propose a variety of executive strategies and explore their consequences, such as what is actually required to coordinate multiple arbitrary tasks. The currently popular application of an oversimplified computer metaphor grossly underestimates the sophistication of both computers and humans. In contrast, the ease of theorizing about cognitive control with EPIC suggests that a good strategy for the cognitive architecture community would be to keep the control characteristics of architectures open as more is learned about complex task performance. In fact, the concepts used in modern computer systems should place a *lower* bound on the mechanisms we consider for human cognitive control processes.

References

Baddeley, A.D., & Logie, R.H.. (1999). Working memory: The multiple-component model. In A. Miyake & P. Shah (Eds.), Models of working memory: Mechanisms of active maintenance and executive control. New York: Cambridge University Press. 28-61.

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.

Bower, G. H., & Hilgard, E. R. (1981). *Theories of Learning*. Fifth Edition. Englewood Cliffs, NJ: Prentice-Hall.

Card, S. K., Moran, T. P., & Newell, A. (1983). *The psychology of human-computer interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Ericsson, K.A., & Kintsch, W. (1995). Long-term working memory. *Psychological Review*, 102(2), 211-245.

Ericsson, K.A., & Delaney, P.F. (1999). Long-term working memory as an alternative to capacity models of working memory in everyday skilled performance. In A. Miyake & P. Shah (Eds.), *Models of working memory: Mechanisms of active maintenance and executive control*. New York: Cambridge Univerity Press. 257-297.

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

John, B. E., & Kieras, D. E. (1996). The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction*, **3**, 320-351.

Kieras, D. E. (1988). Towards a practical GOMS model methodology for user interface design. In M. Helander (Ed.), *Handbook of Human–Computer Interaction* (pp. 135–158). Amsterdam: North–Holland Elsevier.

Kieras, D. E. (1997). A Guide to GOMS model usability evaluation using NGOMSL. In M. Helander, T. Landauer, and P. Prabhu (Eds.), *Handbook of human-computer interaction.* (Second Edition). Amsterdam: North-Holland. 733-766.

Kieras, D.E. (2004). GOMS models and task analysis. In D.Diaper & N.A. Stanton (Eds.), *The handbook of task analysis for human-computer interaction*. Mahwah, New Jersey: Lawrence Erlbaum Associates. 83-116.

Kieras, D.E. (2004). *EPIC Architecture Principles of Operation*. Document available via anonymous ftp at ftp://www.eecs.umich.edu/people/kieras/EPICtutorial/EPICPrinOp.pdf

Kieras, D.E., & Meyer, D.E. (1995). Predicting performance in dual-task tracking and decision making with EPIC computational models. *Proceedings of the First International Symposium on Command and Control Research and Technology*, National Defense University, Washington, D.C., June 19-22. 314-325.

Kieras, D. & Meyer, D.E. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction.*, **12**, 391-438.

Kieras, D. E., & Meyer, D. E. (2000). The role of cognitive task analysis in the application of predictive models of human performance. In J. M. C. Schraagen, S. E. Chipman, & V. L. Shalin (Eds.), *Cognitive task analysis*. Mahwah, NJ: Lawrence Erlbaum, 2000. 237-260.

Kieras, D., Meyer, D., & Ballas, J. (2001). Towards demystification of direct manipulation: Cognitive modeling charts the gulf of execution. *Proceedings of the CHI 2001 Conference on Human Factors in Computing Systems*. New York, ACM. Pp. 128 – 135.

Kieras, D. E., Meyer, D. E., Ballas, J. A., & Lauber, E. J. (2000). Modern computational perspectives on executive mental control: Where to from here? In S. Monsell & J. Driver (Eds.), *Control of cognitive processes: Attention and performance XVIII* (pp. 681-712). Cambridge, MA: M.I.T. Press.

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.

Kieras, D. E., & Polson, P. G. (1985). An approach to the formal analysis of user complexity. *International Journal of Man-Machine Studies*, **22**, 365-394.

Kieras, D.E., Wood, S.D., & Meyer, D.E. (1997). Predictive engineering models based on the EPIC architecture for a multimodal high-performance human-computer interaction task. ACM *Transactions on Computer-Human Interaction*.**4**, 230-275.

Meyer, D. E., & Kieras, D. E. (1997). A computational theory of executive cognitive processes and multiple-task performance: Part 1. Basic mechanisms. *Psychological Review*, **104**, 3-65.

Meyer, D. E., & Kieras, D. E. (1997). A computational theory of executive control processes and human multiple-task performance: Part 2. Accounts of Psychological Refractory-Period Phenomena. *Psychological Review*. **104**, 749-791.

Meyer, D. E., & Kieras, D. E. (1999). Precis to a practical unified theory of cognition and action: Some lessons from computational modeling of human multiple-task performance. In D. Gopher & A. Koriat (Eds.), *Attention and Performance XVII*.(pp. 15-88) Cambridge, MA: M.I.T. Press.

Meyer, D. E., Kieras, D. E., Schumacher, E. H., Fencsik, D., & Glass, J. M. B. (November, 2001). Prerequisites for virtually perfect time sharing in dual-task performance. Paper presented at the meeting of the Psychonomic Society, Orlando, FL.

Meyer, D. E., Kieras, D. E., Lauber, E., Schumacher, E., Glass, J., Zurbriggen, E., Gmeindl, L., & Apfelblat, D. (1995). Adaptive executive control: Flexible multiple-task performance without pervasive immutable response-selection bottlenecks. *Acta Psychologica*, **90**, 163-190.

Monsell, S., & Driver J. (2000). Banishing the control homunculus. In S. Monsell & J. Driver (Eds.), *Control of cognitive processes: Attention and performance XVIII* (pp. 3-32). Cambridge, MA: M.I.T. Press.

Mueller, S. (2002). The roles of cognitive architecture and recall strategies in performance of the immediate serial recall task. Doctoral dissertation. Ann Arbor, Michigan: University of Michigan.

Mueller, S. T., Seymour, T. L., Kieras, D. E., & Meyer, D. E. (2003). Theoretical implications of articulatory duration, phonological similarity, and phonological complexity effects in verbal working memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 2003, *29*, 1353-1380.

Norman, D. A. (1970). *Models of human memory.* New York: Academic Press.

Norman, D. A. (1976). *Memory and attention* (2nd ed.). New York: Wiley.

Norman, D. A., & Shallice, T. (1986). Attention to action: Willed and automatic control of behavior. In R. J. Davidson, G. E. Schwartz & D. Shapiro (Eds.), *Consciousness and self-regulation*, *Vol. 4*. New York: Plenum Press.

Salvucci, D., Kushleyeva, Y., & Lee, F. (2004). Toward an ACT-R general executive for human multitasking. In *Proceeding of the 2004 International Conference on Cognitive Modeling*, Pittsburg, PA July 30-August 1, 2004.

Schumacher, E. H., Lauber, E. J., Glass, J. M. B., Zurbriggen, E. L., Gmeindl, L., Kieras, D. E., & Meyer, D. E. (1999). Concurrent response-selection processes in dual-task performance: Evidence for adaptive executive control of task scheduling. *Journal of Experimental Psychology: Human Perception and Performance*, 1999, *25*, 791-814.

Schumacher, E. H., Seymour, T. L., Glass, J. M., Fencsik, D., Lauber, E. J., Kieras, D. E., & Meyer, D. E. (2001). Virtually perfect time-sharing in dual-task performance: Uncorking the central cognitive bottleneck. *Psychological Science*, 2001, *12*, 101-108.

Stallings, W. (1998). Operating systems: Internals and design principles (3rd Edition). Upper Saddle River, NJ: Prentice Hall.

Tucker, A. (Ed.) (2004). *The Computer Science and Engineering Handbook (2nd Ed)*. Boca Raton, CRC Inc. pp. 46-1 - 46-25.

Yantis, S. & Jonides, J. (1990). Abrupt visual onsets and selective attention: Voluntary vs. automatic allocation. *Journal of Experimental Psychology: Human Perception and Performance, 16,* 121-134.