Fidelity Issues in Cognitive Architectures for HCI Modeling: Be Careful What You Wish For[1]

David E. Kieras
University of Michigan
Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109
kieras@eecs.umich.edu

Abstract
The goal of research to develop cognitive architectures is to increase the accuracy and the amount of detail of the mechanisms and phenomena represented in the architecture. However, in practical modeling for interface design, more approximate and coarser-grained architectures can be not just easier to use, but also avoid not-yet-resolved scientific puzzles that distract from the practical modeler's goals. Two examples are given for how high-fidelity issues can be avoided or simplified for practical modeling.

1 Introduction

By *fidelity*, I mean the scientific accuracy, detail, and completeness of a cognitive architecture in representing psychological mechanisms. You would think that the higher fidelity the architecture, the better - the more scientifically complete, the more accurate will be our models for use in design. But scientific completeness and practical utility don't come hand in hand if the science base is seriously incomplete in areas important to a particular modeling application. In fact, it is a mistake to pursue a hi-fidelity approach to solving a practical design problem when the relevant science is incomplete. Rather than staying on track to solve the design problem, such an effort will get bogged down in research questions or puzzles almost immediately. "Guestimates" and speculative solutions to these puzzles may seriously distort the model predictions, but in a high-fidelity architecture, these mis-solutions will tend to be less obvious because they are buried in all the details. If a low-fidelity architecture model will address the design problem adequately well, why take the trouble and the risk? Simply by virtue of its use, the low-fidelity architecture will make it clear that the model is only approximate.

In my own work, I've been developing two modeling architectures: a low-fidelity architecture based on GOMS models called GLEAN (Kieras, Wood, Abotel, & Hornof, 1995; Kieras, 1999), and a high-fidelity architecture based on a production system surrounded by elaborate perceptual and motor processors, called EPIC (Kieras & Meyer, 1997). Each can be thought of as souped-up versions of the Model Human Processor (Card, Moran, & Newell, 1983), implemented as running and programmable simulation systems, with considerably more detail and specificity than in the original Model Human Processor proposal. Because of space and formatting limitations, these architectures will be described only minimally - even the conventional diagrams won't be shown, and the examples of model code will be kept to a minimum. See the references for more information about each one; however, the newer aspects of GLEAN will receive more attention here.

I will illustrate the fidelity problem in two areas: The first is the visual search involved in interacting with an interface, more specifically, locating and pushing buttons in a GUI. The high-fidelity architecture involves opening a huge can of worms, making the lower fidelity solution very attractive for practical work.

The second area is cognitive parallelism, i.e. having more than one thread or stream of cognitive activity underway at a time. Multitask performance is an obvious case where cognitive parallelism might be involved, but there are several aspects of nominally single-task performance where such parallelism might be involved. The high-fidelity EPIC architecture allows a theoretically elegant and comprehensive approach to representing cognitive parallelism, but it appears at this time that the most important forms of parallelism can be represented well within the low-fidelity GLEAN architecture.

[1] Kieras, D. (2005). Fidelity issues in cognitive architectures for HCI modeling: Be careful what you wish for. Paper presented at the 11th International Conference on Human Computer Interaction (HCII 2005). Las Vegas, July 22-27. Proceedings published as CD-ROM.

2 Visual Search for Buttons

2.1 EPIC Model for Clicking on a Button

EPIC is a production-rule cognitive processor surrounded by perceptual-motor processors that run in parallel, interacting with the cognitive processor and with each other. The visual processor comprises a whole subsystem of storage systems and processors, which interact with an ocular motor system that produces both voluntary and involuntary eye movements. While some properties of this system are well-understood and long-established, we continue to have a very incomplete picture of the visual system. For an excellent recent integrative overview, see Findlay and Gilchrist (2003), who point out that a key feature of human vision is that the visual field is not homogenous in acuity, but it has not been adequately appreciated that eye movements to bring things into foveal view can be accurately guided by ample information available outside the fovea. Furthermore, the mechanisms involved such visual search appear to be far more efficient and capable than normally portrayed in conventional cognitive psychology wisdom. Unfortunately, this interesting picture is seriously incomplete and full of problems. The newer results call for discarding or at least reconsidering a variety of long-standing beliefs in mainstream cognitive psychology ranging from the nature of visual attention to the details of saccade programming. Furthermore, while acuity is known to vary with the perceptual property (e.g. color), eccentricity, size, and crowding, the form and parameters of the acuity functions are complex and incompletely understood.

The current EPIC architecture is compatible with these advanced ideas about the visual system and visual search, and work is underway to include them. But making use of this rich representation of the visual and oculomotor systems is not easy; this will be illustrated with an example set of production rules from an EPIC model for a complex task whose interface is under design. These rules follow some programming conventions for translating GOMS methods into EPIC production rules, so the rules for finding and clicking on a specified button can be termed the method for the click-on-button goal.

The basic rationale for this method is that the user knows the general area where the button is to be found, and so eye can be immediately positioned there instead of searching the entire display. The button is color coded to allow recognition of a candidate visual object over a larger area, specified by the acuity function for object color. As specified by the acuity function for text, the candidate object needs to foveated, whereupon the button label can be recognized and compared to the label for the sought-after button. For simplicity, this method assumes that there is only one button that matches the specifications, and the acuity functions allow it to be located and fixated without trial-and-error search.

Figure 1 shows the production rules that invoke (or "call") the push-button method. Each rule is governed by a goal clause in its condition and a step clause that indicates which step in the method is current. The first rule moves the eye to a "named location" where the buttons are known to appear, and loads working memory with the "parameters" of the method - these are "Tag" memory items for the color and label of the desired button. The rule sets up the subgoal of clicking on a button by adding a goal clause to memory, and finally disables itself by removing the step clause that triggered it, and enables the next step by adding a step clause. The next rule waits for the subgoal to be accomplished (signaled by the disappearance of the subgoal from working memory) and then it sets up the next step in the calling method.

```
ModConstraint_3
If( (Goal Modify Constraint)
    (Step ModConstraint Clickon Button)
    (Motor Manual Modality Free)
    (Motor Ocular Modality Free))
Then((Send_to_motor Ocular Perform Move Button_area_location)
    (Add (Tag Click_on_Button Teal Color))
    (Add (Tag Click_on_Button Modify Label))
    (Add (Goal Click_on Button))
    (Delete (Step ModConstraint Clickon Button))
    (Add (Step ModConstraint Waitfor Button_done))))

(ModConstraint_4
If( (Goal Modify Constraint)
    (Step ModConstraint Waitfor Button_done)
    (Not (Goal Click_on Button)))
Then((Delete (Step ModConstraint Waitfor Button_done))
    (Add (Step ModConstraint Return_with Goal_Accomplished))))
```

Figure 1. Example method call production rules.

Figure 2 shows the rules for clicking on a button using the specified color and label. The first is a start-up rule that enables the first step in the method (disabling itself in the process). The second waits for any previous eye movement to be completed and for the color and shape of any visual object matching the parameters to become available; then it will fire and move the eye to that object, picking one at random if there is more than one. It makes a note (with the Tag item) of which object was chosen.

The next rule in the sequence will fire when the text property of the chosen object becomes available and matches the label parameter. The rule will then command the manual processor to prepare and execute a point-and-click compound movement to the object. The final rule in the method waits for the manual movement to be complete, and then cleans up memory and removes the goal clause to signal that the goal has been satisfied. The flow of control then returns to the calling method which has been waiting for this goal-removal event.

To generalize this method so that it could find a button without a distinctive color would require at least half again as many more rules to implement an iterative visual search process: pick a button, look at it, check the label, repeat if not right. Similar searches have been modeled in EPIC (e.g., Hornof, 2001, 2004).

Here is the point of the example: Notice how many assumptions have to be made to implement clicking on a button. From a scientific point of view, the situation is excellent: nothing is more desirable than to make explicit the factors involved in a task. But from a practical point of view, the fact that the time requirements of the search depend heavily on what is currently a set of free visual acuity parameters is a serious problem. In addition, it should be clear that a variety of additional assumptions about the search strategy are possible: If it is reasonable to say the user knows the general area where the button can be found, couldn't it also be reasonable that the user knows exactly where the button is and so won't have to search for it? If neither of these is true, then the whole display would need

```
(Click_on_button_MFG
If( (Goal Click_on Button)
    (Not (Step Click_on_Button ??? ???)))
Then((Add (Step Click_on_Button Find Button))))

(Click_on_button_find_button
If( (Goal Click_on Button)
    (Step Click_on_Button Find Button)
    (Tag Click_on_Button ?color Color)
    (Visual ?button Color ?color)
    (Visual ?button Shape Button)    // special shape for a button
    (Motor Ocular Modality Free)
    (Randomly_choose_one))
Then((Add (Tag Click_on_Button ?button Button))
    (Send_to_motor Ocular Perform Move ?button)
    (Delete (Step Click_on_Button Find Button))
    (Add (Step Click_on_Button Check_and_click Button))))

(Click_on_button_check_and_click
If( (Goal Click_on Button)
    (Step Click_on_Button Check_and_click Button)
    (Tag Click_on_Button ?button Button)
    (Tag Click_on_Button ?label Label)
    (Visual ?button Text ?label)
    (Motor Manual Processor Free))
Then((Send_to_motor Manual Perform Click_on ?button)
    (Delete (Step Click_on_Button Check_and_click Button))
    (Add (Step Click_on_Button Return_with Goal_accomplished))))

(Click_on_button_RGA
If( (Goal Click_on Button)
    (Step Click_on_Button Return_with Goal_accomplished)
    (Tag Click_on_Button ?x ?y)
    (Motor Manual Modality Free))
Then((Delete (Tag Click_on_Button ?x ?y))
    (Delete (Step Click_on_Button Return_with Goal_accomplished))
    (Delete (Goal Click_on Button))))
```

Figure 2. Production rules for locating and clicking on a button.

to be searched, for which there are many different strategies. Depending on the choices, the time predicted by the model for this elementary interaction task will cover an extremely wide range. I wish I could say that we have results that suggest at least an acceptable "default" set of acuity functions and search strategy - but we don't at this time. This means that the practical application of this high-fidelity model is complex and difficult: modeling the simple operation of "click on the X button" becomes a serious research enterprise.

2.2 GLEAN Model for Clicking on a Button

GLEAN is a simplified cognitive architecture that is very similar to the EPIC architecture. The major differences are that the GLEAN perceptual and motor processors implement the Keystroke-Level Model operators (see John & Kieras, 1996a,b), and so are much more approximate and larger-grain than in EPIC, and the cognitive processor is controlled by GOMS models written in a procedural language (called GOMSL) instead of production rules, as in EPIC. Although the GLEAN architecture is considerably simpler than EPIC, GLEAN has been successfully applied to much more complex tasks than EPIC or most other cognitive architectures, such as the modeling of human teams in a Navy anti-air warfare task setting (Kieras & Santoro, 2004); the basic approach was elegant and simple: we modeled a team of interacting humans with a team of interacting models.

Figure 3 shows the GOMSL for the process of clicking on a specified button, taken from the Kieras & Santoro models. The first method calls the second. By design, GOMSL can be read almost like an ordinary procedural programming language. The first method is called when it is necessary to close a certain window. The first line names the method goal; the first step "calls" the submethod by setting up the goal to Click_on Button, and passes along a parameter, the label of the desired button. After the subgoal has been accomplished, control returns to the second step, which notes that the Close goal has been accomplished, and control returns to whichever method called for accomplishing the Close goal.

The second method actually carries out the process of clicking on a specified button. The first line, the Method statement, names the method goal and assigns a local name, <click_on_button_label>, to the parameter information deposited in working memory when the method goal was set up in the calling method. The first step contains a keystroke-level *Look_for* operator which is assumed to take an amount of time based on the Keystroke-Level Model Mental (**M**) operator (e.g., 1.2 s; see John & Kieras, 1996a, b). The name of the visual object matching the label and Type specifications is deposited in working memory as <click_on_button>. For simplicity, this method assumes that the Look_for operator will always succeed, so there is no check on the result. The second step executes after the first is complete, and executes a mouse *Point_to* operator to the named visual object. If the screen location and size of this object have been specified, a variation of Fitts' law is used to calculate the movement time based on the current cursor position; if not, it defaults to the conventional Keystroke-Level Model time (1.1 s). The third step executes when the movement is done, and removes the temporary memory items and causes the method to terminate with its goal accomplished.

The extreme simplicity of the GLEAN model is a strong contrast to the EPIC model, but how accurate is it? In line with the long track record of GOMS models (John & Kieras, 1996a, b), the Kieras and Santoro models are usefully accurate on several metrics of performance of whole tasks that involve many button-pushes and other activities. At the smaller grain of this example, the Click_on Button method will either be roughly correct or seriously incorrect when applied to a specific task context, button, interface design, and user population. But how could the accuracy be determined without collecting a lot of detailed data on the performance time with the actual interface with actual users? If such data is available, it can be used to produce a better estimate for the time taken by

```
Method_for_goal: Close Track_data_window
Step 1. Accomplish_goal: Click_on Button using "Close".
Step 2. Return_with_goal_accomplished.

Method_for_goal: Click_on Button using <click_on_button_label>
Step 1. Look_for_object_whose Label is <click_on_button_label>,
    and Type is Button and_store_under <click_on_button>.
Step 2. Point_to <click_on_button>.
Step 3. Click Left_mouse_button.
Step 4. Delete <click_on_button>; Return_with_goal_accomplished.
```

Figure 3. Methods for calling and executing clicking on a button.

the *Look_for* operator, which *de facto* takes all of the visual acuity factors and search strategies into account. However, why would one collect such data? Of course, one reason for collecting detailed data with an existing interface would be to calibrate a model for the existing interface and then apply a variation of the model to a new, but basically similar, interface design. But if it is possible to collect data adequate to validate in detail a model of a single design, then it should be possible to perform normal full-scale usability tests. However, if detailed data can't be collected, either for economic reasons or because the system under design has not even been prototyped, then the simple fact of the matter is that the actual accuracy of the GLEAN (or GOMS) model is unknown; our previous experience suggests only that it will be in the "right ballpark" - but maybe this is good enough.

In this situation, rather than get bogged down in the details of the high-fidelity EPIC model, which involves hard-to-document guesses about parameters and hard-to-implement strategies, we can apply the low-fidelity GOMS model that has only very simple and obvious assumptions - a *Look_for* takes an **M**, and that's all there is to it. It might be inaccurate, but if we don't have the data to actually tell, at least we haven't wasted a lot of time constructing a high-fidelity model that might be just as inaccurate.

Thus, at this point in time, the scientific situation for visual search is so seriously underdeveloped that trying to capture it in a high-fidelity model is a research project, not a design project. If a design prediction is needed, the low-fidelity approach is the clear winner.

3 Modelling Cognitive Parallelism

3.1 Multiple Threads in EPIC

In addition to representing the perceptual-motor constraints on performance, a major purpose of the EPIC architecture is represent human multitasking and executive processes. The key to the theoretical approach is massive parallelism in the production system: Specifically, any number of rules can fire on a single production-rule cycle, but this does not mean that EPIC's cognitive processor can perform unlimited computations. This is because EPIC's production rules can be triggered only by a limited amount of information from the perceptual systems, such as the limits imposed by visual acuity, and the limited amount of information and processing rates in the working memory systems, such as a phonological-loop memory (Kieras, Meyer, Mueller, & Seymor, 1999). What the rules can do when they fire is limited by the motor processors, which can only prepare one distinct movement at a time. So being able to fire many rules simultaneously does not give EPIC infinite computational power. Rather what it does is enable a production-rule programming paradigm closely related to long-popular multithreaded computer programming techniques (see Kieras, Meyer, Ballas, & Lauber, 2000 for more discussion).

Figure 4 is a schematic illustration of how cognitive parallelism is implemented in EPIC production rules; only goal and step clauses are shown in the rules. The upper set of rules in the figure shows how more than one sub goal can be set up and simultaneously executed; it is a generalization of the submethod call shown in Figure 1. The first rule adds the clauses for both goals and enables the second rule, which waits for both goal clauses to disappear. In a more elaborate case, additional steps in the calling method can be done before waiting for either subgoal to be

```
(StartParallelSubGoals
If ((Goal X) (Step A))
Then ((Add (Goal Y))(Add (Goal Z)(Delete (Step A))(Add (Step B))))

(ParallelSubGoalsFinished
If ((Goal X)(Step B) (Not(Goal Y)(Not(Goal Z)))
Then ((Delete (Step B))(Add (Step C)))

(StartParallelSteps
If ((Goal X)(Step A))
Then ((Delete (Step A))(Add (Step B1))(Add (Step B2))))

(ParallelStepB1
If ((Goal X)(Step B1))
Then ((Delete (Step B1))(Add (Step C1))))

(ParallelStepB2
If ((Goal X)(Step B2))
Then ((Delete (Step B2))(Add (Step C2))))
```

Figure 4. Schematic illustration of EPIC parallel execution programming.

accomplished. The lower set of rules in the figure shows parallel steps within a method: The first rule adds two step clauses, each of which enables a distinct next step; such a step thread can either be merged back into another step thread when one of its rules waits for the completion of a step, or else it can simply terminate by deleting its last step without enabling another.

What keeps multiple threads from conflicting with each other? In EPIC, as in computer programming, threads can conflict only if they try to simultaneously modify a datum or control a resource (such as an output device). This requires careful programming, both in EPIC and in computer programming.

What is the advantage of multiple threads? Oddly enough, it is not that while one line of processing is waiting for something to be completed, another line of processing can be making progress in the task. It is possible to obtain such interleaved operation without multiple threads - the programming is just very difficult and tedious. Rather the advantage is that when programming with multiple threads, only the points of conflict need to be worked out carefully - the rest of the time, the programming can be done as if no other processes were involved. In other words, multithreaded programming was developed in order to allow greater throughput with relatively easy programming.

An example of multithread programming in EPIC is the models exploring direct manipulation in a complex dual task (Kieras, Ballas, & Meyer 2001). The task involved selecting, examining, and then classifying "blips" on a radar display. The visual search for the next blip to process can be overlapped with the completion of the classification response for the previous blip. However, the extent of overlapping is limited by whether the classification response requires visual guidance (as in a touchscreen) or not (as in a keypad). The multithreaded capability of EPIC allowed this overlap to be represented relatively easily with production rule threads that executed when their required resources were available and the task constraints had been satisfied.

Another example of how the multithreaded capability is useful in a complex task is the ease of implementing "demons" - processes that constantly "watch" for a particular event and initiate processing when it happens. The first rule in Figure 5 is a trivial schematic example of such a rule.

Note how this rule is not governed by any goal or step clause. If a visual object appears whose color is red, the rule will fire, no matter what else is going on. The second example rule in Figure 5 is based on the models in Kieras, Ballas, and Meyer (2001). If a blip is being processed, and the current blip disappears from the screen, the rule fires, halts the processing and starts up a clean-up method which will eventually result in a new blip being selected for processing. Demons in EPIC are simple for two reasons: the demon rule can fire at any time, regardless of which other rules are firing, and the demon rule actions can directly modify the flow of control through other rules by manipulating goal clauses. Thus, in keeping with multiple threading as a simplification, rather than complicated normal methods containing repeated checks for the continued existence of the current blip, this demon simply watches for this unusual situation and handles it with special routines if it occurs. This illustrates the great convenience of cognitive parallelism in constructing models for complex tasks.

As in multithreaded computer programming, multithreaded production rule programming is extremely convenient in certain situations, but is also technically difficult. The programmer has to watch out for various conditions to protect against conflicts in modifying data or controlling resources; Kieras et al. (2000) discuss these issues in the context of so-called executive functions in humans, using computer operating systems as a source of relevant concepts.

```
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 5. Examples of demon rules in EPIC.

3.2 Multiple Threads in GLEAN

My claim is that in most normal HCI tasks the following cognitive parallelism situations are the most important:

- Overlapping of motor behavior, so that the eyes, hands, and voice can be controlled by separate methods as independently as the task allows.
- Watching for high-priority events that interrupt ongoing activity, cause special high-priority execution of special methods, and then either terminate or resume the interrupted activity. One example is task events such as appearance or disappearance of blips in a radar operation task, but a more general case is the detection of an error.

In terms of threading control structures, these are a subset of the possibilities that EPIC's production rules allow, which suggests that a simple representation in GOMSL and GLEAN should be possible. In fact, GLEAN has long incorporated some of this capability (see Kieras and Santoro, 2004) and additional multithreading capabilities are now in the current experimental GLEAN4.

3.2.1 Multiple GOMS threads in GLEAN4

GLEAN4 is a version of GLEAN in which the cognitive processor executes GOMS methods in possibly multiple threads. Applying elementary computer operating systems concepts, each thread has a top-level goal and its own goal stack, but all memory is shared (as in EPIC's production system). Thus the basic cognitive parallelism available is that multiple methods can be simultaneously executed, not multiple steps within a single method.

Each GOMSL step that contains a perceptual or motor operator requires control of the corresponding resource (e.g. the manual motor processor) before it can be executed. If that resource is busy, then the step is kept waiting until the resource is free. If two threads need the same resource at the same time, then the higher priority thread gets it. If there is no conflict in resources, the next step is executed for both threads, allow the processing to continue simultaneously in both.

There is always a main thread where execution starts. The modeler can specify interrupt rules, which means that at start up, a highest-priority thread for the interrupt rules is started along with the main thread. On every cycle, the interrupt rules are tested, and the actions executed if one fires, typically starting the execution of submethods. These submethods can use any resources and run at higher priority than all other processes; once they complete, the suspended threads are resumed. The work summarized in Kieras and Santoro (2004) led very early to the need for interrupt capabilities to detect and handle auditory speech inputs from other team members, as well as high-salience events happening on the display.

Figure 6 shows a simple demonstration of GLEAN4 thread programming. The main thread starts with the method for Perform Demo, the designated starting goal. It starts two subthreads, running at background priority, one for Do ABC and the other for Do XYZ. These two methods, running as separate threads, automatically interleave their use of the manual and vocal processors, with keystrokes going ahead in one method while speech is taking place in the other. In the meantime, as a demonstration, the main thread "spins" waiting for a <signal> item to appear in working memory. On each 50 ms cycle, the interrupt rule is tested. When both submethods have completed and deposited their status items in memory, the interrupt rule fires and starts a shutdown method that runs at interrupt priority. After generating some speech and cleaning up, the interrupt method deposits the signal that the main thread is watching for, and terminates. The main thread then terminates and the simulation is complete. The interrupt rule is not really needed to produce the demonstration behavior, but is included just for illustration.

GLEAN4 thus automatically allocates resources to threads; each step in a process waits no longer than its priority and dependencies require. Limiting multithreading to these simple forms, as small extensions to GOMSL, means that the modeler can incorporate important form of multitasking and parallelism easily.

3.2.2 Error Handling in GLEAN4

The basic multithreading capability enabled us to work with developing an approach to adding mechanisms for error detection and recovery to GLEAN. This is work done with Scott Wood, following up on his earlier theoretical work (Wood, 2000; Wood & Kieras, 2002) on extending GOMS to handle errors based on exception-handling concepts from programming languages, but assuming a more general model than normally implemented. In essence, his work followed up on the Card, Moran, and Newell (1983) discussion of errors in GOMS. Once an error is committed, handing it becomes a goal, for which the user should have good methods. The technical difficulties come from how error recovery will involve changes to the current goal and method execution state. Wood's insight was that error recovery involves mechanisms like exception-handling that are able to "stand outside" the failed

```
Define_model: "ThreadDemo"
Starting_goal is Perform Demo.

Interrupt_rules
If <ABC_status> is Done, <VWX_status> is Done,
    Then Accomplish_goal: Shutdown Demo.
Method_for_goal: Shutdown Demo
Step I1. Speak "Game Over!".
Step I2. Delete <ABC_status>; Delete <VWX_status>; Store Stop under <Signal>;
    Return_with_goal_accomplished.

Method_for_goal: Perform Demo
Step Top1. Also_accomplish_goal: Do ABC as Background.
Step Top2. Also_accomplish_goal: Do VWX as Background.
Step Top3. Decide: If <Signal> is Stop Then Return_with_goal_accomplished;
    Else Goto Top3.

Method_for_goal: Do ABC
Step A. Speak "I am doing step A".
Step B. Keystroke B.
Step C. Keystroke C.
Step D. Point_to D.
Step E. Store Done under <ABC_status>; Return_with_goal_accomplished.

Method_for_goal: Do VWX
Step V. Keystroke V.
Step W. Speak "I am doing step W".
Step X. Keystroke X.
Step Y. Keystroke Y.
Step Z. Store Done under <VWX_status>; Return_with_goal_accomplished.
```

Figure 6. Demonstration of multiple threads in GLEAN4.

method tree in order to decide how to recover, and how to carry it out. The multithread capability makes this relatively easy to implement.

In Wood's formulation, each method can specify a goal for handling errors that occur during execution of this method or its submethods. A method step or operator can raise an exception if it detects an error condition, whereupon the current thread is suspended, an error thread is created, and control transferred to it with information in working memory identifying the failed method, step, and cause. This thread begins executing the method for the error goal specified by lowest-level method underway at the time of the error. The error method can call submethods to interact with the interface to bring the system to an appropriate state, and decide how to resume processing; the options include restarting the failed method, retrying the failed step, aborting the failed method and restarting a higher level method.

A brief example using an excerpt from Wood's "Webstock" appears in Figure 7. The Webstock application worked well at producing human errors in the context of a stock-trading web application (Wood & Kieras, 2002), and served as a good modeling testbed. In this example, an earlier error in following a link would be detected when visual search fails to find the expected item on the screen. However the method that detects the error can be called from more than one place in the model, and so the error recovery method needs to be chosen depending on the context of the error.

The top-level method in the example looks up information about a stock using first the name of the page to go to, and then the first letter of a stock symbol. For brevity, the remainder of the method is not shown. The Follow Link method looks for a link of the specified name, and then points to it. If the page is incorrect, the proper link visual object will be missing, and the Look_for operator will store an *Absent* result in the <link_object> memory slot. The GLEAN4 Point_to operator will raise an error exception if its argument is invalid, so at this point in the process, the error is detected. The Follow Link method is suspended, and the error thread begins execution to accomplish the goal of Handle Lookup_Error. The recovery method examines the current page and decides which method to restart and at what point to restart it. The abort_and_restart option is a last-ditch "start all over" choice.

Notice how, consistent with how exceptions benefit programming languages, the non-error flow of control is completely normal and not cluttered with error handling code. Rather, problems are detected within the operators or where appropriate in the methods, and then handled via a separate flow of control that executes methods dedicated

```
Method_for_goal: Do Lookup using <page_name>, <stock_letter>
On_error: Handle Lookup_Error
Step DLFL1. Accomplish_goal: Follow Link using <page_name>.
Step DLFL2. Accomplish_goal: Follow Link using <stock_letter
...

Method_for_goal: Follow Link using <link_name>
Step FL1. Look_for_object_whose Type is Link, and Label is <link_name>
                  and_store_under <link_object>.
Step FL2. Point_to <link_object>. // raises error exception if target is
Absent
Step FL3. Click B1.
Step FL5. Wait_for_visual_object_whose Event_type is New, and Type is
"Page" and_store_under <current_page>.
Step FL6. Delete <link_object>; RGA.

/* Error methods */
Method_for_goal: Handle Lookup_Error
Step HLE1. Speak "Oops".
Step HLE2. Look_at <current_page>.
Step HLEDecide. Decide:
    If Label of <current_page> is "Start Page" Then
          Accomplish_goal: Recover_from Page_name_error;
    If Label of <current_page> is "Lookup Page" Then
          Accomplish_goal: Recover_from Stock_letter_error;
    Else Abort_and_restart.
Step Done. RGA.

Method_for_goal: Recover_from Page_name_error
Step 1. Accomplish_goal: Get_back_to StartPage.
Step 2. Store "Lookup" under <page_name>;
    Resume <Exception_current_thread> accomplishing_goal Do Lookup at
DLFL1.
Step 3. Return_with_goal_accomplished.

Method_for_goal: Recover_from Stock_letter_error
Step 1. Store "M" under <stock_letter>; // a simple stand-in
    Resume <Exception_current_thread> accomplishing_goal Do Lookup at
DLFL2.
Step 2. Return_with_goal_accomplished.
```

Figure 7. Example error handling using GLEAN4 in Wood's Webstock.

to error recovery. Now that we have a working first version of a executable GOMS model that incorporates error detection and recovery, we are able to address the substantial challenge of developing models for when errors occur, characterizing the most psychological realistic recovery methods, and addressing the practical concerns of helping developers make their systems robust against human errors.

Thus, while a high-fidelity architecture can represent fully the many possibilities for cognitive parallelism in multitasking, the forms that are most important for practical modeling of user interfaces for complex tasks can be represented in an easy-to-use form in a low-fidelity architecture. The low-fidelity approach wins again.

4 Conclusion

The future of cognitive architecture modeling in HCI and UI development requires solving a very large number of both theoretical and practical problems. The trick is to avoid have to solve them all before we can do anything useful. The experience gained in applying limited and incomplete low-fidelity models to real problems can inform and guide the development of high-fidelity models, which can then inform future low-fidelity models. By deliberately pursuing both high and low fidelity solutions, and constantly using them to inform each other, we can be practically useful while we are trying to get scientifically more accurate.

References

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

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

Hornof, A. J. (2004). Cognitive strategies for the visual search of hierarchical computer displays. *Human-Computer Interaction*, 19(3), 183-223.

Hornof, A. J. (2001). Visual search and mouse pointing in labeled versus unlabeled two-dimensional visual hierarchies. *ACM Transactions on Computer-Human Interaction*, 8(3), 171-197.

John, B. E., & Kieras, D. E. (1996). Using GOMS for user interface design and evaluation: Which technique? *ACM Transactions on Computer-Human Interaction*, **3**, 287-319.

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. (1999). *A Guide to GOMS Model Usability Evaluation using GOMSL and GLEAN3*. Document and software available via anonymous ftp at ftp://www.eecs.umich.edu/people/kieras

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., 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. & Santoro, T.P. (2004). Computational GOMS modeling of a complex team task: Lessons learned. In *Proceedings of CHI 2004: Human Factors in Computing Systems*. New York: ACM, Inc.

Kieras, D.E., Wood, S.D., Abotel, K., & Hornof, A. (1995). GLEAN: A computer-based tool for rapid GOMS Model Usability Evaluation of User Interface Designs. In *Proceeding of UIST*, 1995, Pittsburgh, PA, USA, November 14-17, 1995. New York: ACM. pp. 91-100.

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.

Wood, S. D. (2000). Extending GOMS to human error and applying it to error-tolerant design. Doctoral dissertation, University of Michigan.

Wood, S.D., & Kieras, D.E. (2002). Modeling human error for experimentation, training, and error-tolerant design. *Proceedings of the Interservice/Industry Training, Simulation, and Education Conference*, pp. 1075-1085, Orlando, FL. December, 2002.