NORTHWESTERN UNIVERSITY

Simulating Human Performance in Complex, Dynamic Environments

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree of

DOCTOR OF PHILOSOPHY

Field of Computer Science

by

Michael Alan Freed

EVANSTON, ILLINOIS
June 1998

ABSTRACT

Simulating Human Performance in Complex, Dynamic Environments

Michael Alan Freed

Computer simulation has become an indispensable tool in numerous design-engineering domains. It would be desirable to use this technique to help design human-machine systems. However, simulating the human component of such systems imposes challenges that have not been adequately addressed in currently available human models. First, a useful human model must be able to perform capably in domains of practical interest, many of which are inherently complex, dynamic, and uncertain. Second, models must be able to predict design-relevant aspects of performance such as human error. Third, the effort required to prepare the model for use in a new task domain must not be so great as to make its use economically infeasible. This thesis describes a modeling approach called APEX that address these challenges.

The APEX human operator model is designed to simulate human behavior in domains such as air traffic control. Performing tasks successfully in such domains depends on the ability to manage multiple, sometimes repetitive tasks, using limited resources in complex, dynamic, time-pressured, and partially uncertain environments. To satisfy this capability requirement, the model adapts and extends sketchy planning mechanisms similar to those described by Firby [1989]. The model also incorporates a theory of human error that causes its performance to deviate from correct performance in certain circumstances. This can draw designers' attention to usability problems at an early stage in the design process when fixing the problem poses relatively little cost. APEX also includes means for managing the substantial cost of preparing, running, and analyzing a simulation.

ACKNOWLEDGEMENTS

First, I would like to thank my advisors Larry Birnbaum and Gregg Collins. Larry is the best teacher I have ever had. Through his many ideas, passionate opinions and intellectual energy he has, more than anyone, determined the kind of researcher I am and want to become.

Deliberate, precise, and virtually never wrong, Gregg provided a productive and often entertaining counterpoint to Larry's exuberant style. His insights and incisive comments were invaluable for having rid me of some bad ideas, and for helping me learn to read, write, and think about hard problems.

Thanks also to the other members of my committee. Chris Riesbeck has impressed me with his knowledge and intellectual integrity. I've enjoyed our discussions a great deal. Louis Gomez was kind enough to join my committee at the last moment, never having even met me. And special thanks to Mike Shafto whose enthusiastic discussions, advocacy of my work, and willingness to spend so much of his limited time on my behalf have often made hard work seem a pleasure.

Paul Cohen introduced to me to AI research. He hired me into his lab, convinced me that I'd rather be a researcher than a lawyer (phew!) and gave me a lot of interesting things to do. Paul can also be credited with (or blamed for) convincing me that research is only fun if one takes on big problems, a view that probably added years to my graduate career but made it possible for me to love the work.

Roger Schank took me with him to Northwestern from Yale. Although our ways parted soon afterwards, I feel grateful to Roger for creating a wonderfully vital research environment. Thanks also for the good advice, great stories, and brilliant ideas.

After several years, I left Northwestern to work at NASA Ames Research Center. I'm indebted to Jim Johnston for hiring me, teaching me about real estate and for countless fascinating discussions. Thanks to Roger Remington for making the Coglab a great place to work, and for enabling me to focus on my research full-time. Thank you especially for believing in me when you weren't sure whether you believed in what I was doing.

Earning a Ph.D. can be a dry and solitary experience without great friends who are also great colleagues. Robin Burke, Bill Ferguson, and Barney Pell have done more than anyone to enrich the experience – i.e. to fill my head with irrelevant ideas and waste my time. Thanks guys. I'm looking forward to more of the same.

Many others have contributed to my education, well-being, and to the completion of this dissertation including Cindy Bernstein, Matt Brand, Kevin Corker, Michael Cox, Eric Domeshek, Danny Edelson, Sean Engelson, Andy Fano, Richard Feifer, Will Fitzgerald, Ken Forbus, Jeremy Frank, Eva Gilboa, Eric Goldstein, Rob Harris, Ann Holum, Suzanne Hupfer, Kemi Jonah, Eric Jones, Kevin Jordan, Alex Kass, Jung Kim, Bruce Krulwich, Chris Lopez, Dick Osgood, Enio Ohmaye, Eric Ruthruff, Mark van Selst, Eric Shafto, Greg Siegle, and Ed Swiss.

I can hardly begin to thank my family enough. Their love, support, and confidence in me have meant far more than they know. Thanks for trying to understand why I didn't want to be one of those other kinds of doctors. Most of all, I want to thank my wife and best friend Diana, whose love, intelligence and good nature make it all worthwhile.

To Steve and Paulette Freed

Contents

Chapter 1
Simulation in Design

1.1 Modeling human performance

Computer simulation has come to play an increasingly central role in the design of many different kinds of devices and systems. For some – automobile engines, airplane wings, and electronic circuits, for example – simulation has long been an integral part of the design engineering process. By allowing engineers to evaluate designs at an early stage in this process, simulation postpones the need for a physical prototype; engineering costs decrease in numerous ways, resulting in improved reliability, greater innovation, faster development time, and lower overall cost of development.

Most systems designed with the help of computer simulation do not include a human component. Systems with a "human in the loop" – i.e. those where the performance of a human agent has a significant effect on the performance of the system as a whole – are difficult to simulate because existing human operator models are usually inadequate. However, exceptions in which simulation has been successfully and usefully applied to the design of a human-machine system (e.g. [John and Kieras, 1994]) indicate unrealized potential.

Some of this unrealized potential can be illustrated with a simple example. Consider the task of withdrawing cash from an automatic teller machine (ATM). Withdrawals from most current ATMs involve a sequence of actions that begin with the user inserting a magnetic card, and end with collecting the requested cash and then retrieving the card. A well-known problem with the use of ATMs is the frequency with which users take their money but forget to retrieve their cards [Rogers and Fisk, 1997].

Newer generations of ATMs avoid this problem by inverting the order of the final two steps, forcing users to retrieve their cards before the machine dispenses the requested money. This way of compensating for human forgetfulness is what Donald Norman [Norman, 1988] calls a “forcing function.” By placing an easily neglected task step on the critical path to achieving a goal, the designer drastically reduces the likelihood of neglect.

Though it is certainly accurate to cite users’ proneness to error as a cause of the lost card problem, the availability of a simple way to circumvent user forgetfulness makes it more useful to blame the ATM designers (or the design process they followed) for their failure to incorporate this fix at the outset. Seen as a failure of design, the ATM example illustrates several important points concerning how simulation might become more useful as an engineering tool for human-machine systems. In particular, 1) a practically useful human simulation model must predict operator error or other design-relevant aspects of human performance; 2) usability problems that are obvious from hindsight will not necessarily be discovered in advance; computer simulation should be used to compensate for difficulties designers face in predictively applying their common-sense knowledge about human performance; and 3) many task domains in which such predictions would be especially useful require human models with diverse and sophisticated functional capabilities.

1.2 Predicting operator error

To add value to a design engineering process, a human operator model must be able to predict aspects of human performance that are important to designers. For instance, such models have been used to predict how quickly a trained operator will be able to carry out a routine procedure [Card et al., 1983; Gray et al., 1993]; how quickly skilled performance will emerge after learning a task [Newell, 1990]; how much workload a task will impose [Corker and Smith, 1993]; whether the anthropometric properties of an interface (e.g. reachability of controls) are human-compatible [Corker and Smith, 1993]; and whether multiple operators will properly cross-check one another’s behavior [MacMillan, 1997].

To predict the ATM lost card problem using simulation would require a model of error-prone human behavior. The importance of predicting operator error at an early design stage has often been discussed in the human modeling literature [Olson and Olson, 1989; Reason, 1990; John and Kieras, 1994], but little progress has been made in constructing an appropriate operator model. This failure of progress stems primarily from a relatively weak scientific understanding of how and why many errors occur.

> "..at this time, research on human errors is still far from providing more than the familiar rough guidelines concerning the prevention of user error. No prediction methodology, regardless of the theoretical approach, has yet been developed and recognized as satisfactory. [John and Kieras, 1994]"

Lacking adequate, scientifically tested theories or error (although see [Kitajima and Polson, 1995; Byrne and Bovair, 1997; Van Lehn, 1990]), it is worth considering whether relatively crude and largely untested theories might still be useful. This kind of theory could prove valuable by, for example, making it possible to predict error rates to within an order of magnitude, or by directing designers' attention to the kinds of circumstances in which errors are especially likely. Everyday knowledge about human performance may offer a valuable starting point for such a theory. Though unable to support detailed or completely reliable predictions, a common-sense theory of human psychology should help draw designers attention to usability problems that, though perhaps obvious from hindsight, might not otherwise be detected until a late stage when correcting the problem is most expensive. The ATM example illustrates this point.

Failing to retrieve one's bank card from an ATM is an instance of a *postcompletion error* [Byrne and Bovair, 1997], an "action slip" [Reason, 1990] in which a person omits a subtask that arises in service of a main task but is not on the critical path to achieving the main task's goal. Such errors are especially likely if subsequent tasks involve moving to a new physical environment since perceptual reminders of the unperformed subtask are unlikely to be present.

Postcompletion errors occur in many human activities. For example, people often fail to replace their gas cap after refueling or to recover the original document after making a photocopy. When told the definition of a postcompletion error, people with no background in human psychology or other field of study relating to human performance immediately recognize it as a common form of error and can often generate examples from their own experience. If such errors are familiar from everyday life, why did ATM designers apparently fail to apply this experience to their engineering task?

Engineering practice may be partly at fault. Engineers have not traditionally been educated about human factors that tend to affect system performance such as fatigue, individual differences in visual function, and people's proneness to make certain kinds of errors. Nor have they traditionally been encouraged to take these factors into account when designing new devices and procedures. Problems of human-machine interaction are considered, when they are considered at all, as cosmetic issues to be resolved by late-stage tweaks to the user-interface or by user training, not as central constraints on system design. Even in systems where human performance is known to be of great importance, the process of detecting and resolving human-machine interaction problems usually occurs at a late stage in design when an advanced prototype has been constructed for usability testing by live operators.

1.3 Everyday knowledge about human performance

Though overly machine-centered engineering practice is an important concern, failures to anticipate usability problems in human-machine systems often arise from another, possibly more fundamental, reason. In particular, thinking concretely about human-machine interactions is an inherently difficult problem that requires overcoming several obstacles.

First, the ability to anticipate common forms of error in *novel* situations and domains depends on abstract knowledge about human proneness to error. Abstract knowledge has generally proven easy for people to use for recognition and explanation, but relatively difficult to use for other cognitive tasks such as problem-solving and interpretation [Anderson, 1982; Gick and Holyoak, 1987]. Thus, even an exceptionally diligent engineer, someone with a knowledge of and explicit intention to eliminate postcompletion errors, might not easily identify ATM card-retrieval as problematic. Based on the description of postcompletion errors given above, identifying card retrieval as a likely source of such errors would require determining that: (1) in many cases, acquiring money will be the user's main goal when interacting with the ATM; (2) the need to retrieve the card will arise in service of this goal but (3) is not on the critical path to achieving it; (4) many users will immediately leave the physical environment of the ATM after achieving their main goal; and so on.

Neither everyday experience or standard design engineering practice prepares an engineer to go through such a thought process. This observation has led to the development of a human factors methodology called Cognitive Walkthrough (CW) [Polson et al., 1992] wherein a designer systematically examines hypothetical scenarios in which a typical user attempts to operate the machine being designed. The CW technique provides a fairly simple (and thus limiting) formalism for describing how users select actions in service of their goals and how they update their situational knowledge in preparation for the next cycle of action. Each stage in the action selection process is associated with a checklist of potential human-machine interaction problems. Employing CW involves using the formalism to methodically step through ("hand simulate") the cognitive and physical activities of the hypothetical user while considering potential problems listed on the checklists.

The effectiveness of CW depends largely on the checklists. For example, in the version of this methodology described in [Polson et al., 1992], whenever the simulated user is to choose and execute an action, questions such as the following are to be considered:

- Is it obvious that the correct action is a possible choice here? If not, what percentage of users might miss it ? (% 0 25 50 75 100)
- Are there other actions that might seem appropriate to some current goal? If so, what are they, and what percentage of users might choose these? (% 0 25 50 75 100)
- Is there anything physically tricky about executing the action? If so, what percentage of users might have trouble? (% 0 25 50 75 100)

Asked in the context of a specific machine state, task state, and user mental state, such questions enable a designer to think concretely about possible design-facilitated user errors and other usability problems. This goes a long way towards overcoming the first obstacle to predicting error at an early design stage, that of applying abstract knowledge about human performance. However, CW falls short in dealing with a second, equally important obstacle: coping with an enormous increase in detail that needs to be kept track of as machine complexity, task duration, and human model sophistication increase in new and more challenging task domains.

CW has been applied to tasks such as call-forwarding [Polson et al., 1992] and operating a mail messaging system [Lewis, 1990], but not to more demanding tasks such as flying a commercial airliner or controlling an automated manufacturing system (though see [Wharton et al., 1992]). The use of a simplified model of human action selection prevents CW from being applied to such tasks, though this might be remedied by a more sophisticated model. Similarly, other aspects of these more challenging domains – longer task durations, more complex and dynamic task environments, an increased number of scenarios that need to be explored – require improvements to the CW approach but are not necessarily at odds with it. However, the use of hand simulation, as opposed to computer simulation, presents an intractable problem.

As discussed, simulating the internal and external activities of the machine and user in great detail makes it possible to use abstract knowledge about human performance. But such detail presents problems for the human designers who need to keep track of it all. As the human model employed becomes more sophisticated, and other measures are taken to scale up to more demanding task domains, the amount of information needed to characterize the state of the simulated world and simulated user increases dramatically.

This proliferation of detail presents the additional difficulty for CW of having to examine (with checklists) not tens or hundreds of events, but tens of thousands or millions. Thus, not only is hand simulation infeasible for scaled up analyses, but so is the use of a person to scrutinize each individual simulation event. If sophisticated human operator models are to be employed, and the more complex and demanding task domains explored, the role of the human analyst must be minimized. Computer simulation must replace the human analyst wherever possible.

1.4 Acting in demanding task environments

Numerous computer models of human behavior have been constructed. For present purposes, it is useful to divide them into two categories. *Psychological models* are constructed primarily to explore or help validate psychological theories. For instance, the EPIC system [Kieras and Meyer, 1994] implements a theory of concurrent "dual-task" execution. SOAR [Laird, 1987; Newell, 1990; Rosenbloom et al, 1991], ACT-R [Anderson, 1990], and CAPS [Just and Carpenter, 1992] represent theories of learning and memory. Such systems can and have been applied to practical problems, but constructing an accurate model of human cognition has been the primary goal.

Engineering models of human behavior, sometimes called "human operator models," are constructed specifically to help achieve practical engineering objectives. Psychological findings are incorporated when they serve these objectives. MIDAS [Corker and Smith, 1993], for example, has been used to assess the legibility of information displays and to predict whether a task's workload requirements will exceed normal human capacity. While predictive accuracy is an important concern in designing such models, the intention to supply useful engineering advice can present other, sometimes conflicting requirements. In particular*, a useful human operator model must be able to act proficiently in the simulated task environment under investigation.*

In many domains, satisfying this requirement entails modeling human capabilities whose underlying mechanisms and limitations are not well-understood. For example, people have some ability to recall an intended action once it is time to act and to choose the best method for achieving the intention based on current situational knowledge, but no detailed, scientifically supported account of these processes exists. In many task domains, such capabilities are exercised as a part of normal behavior and make it possible to achieve normal levels of performance. Operator models must incorporate a broad range of human capabilities if they are to be used to analyze more complex and more diverse task environments.

Constructing a useful human operator model requires addressing two challenges. The first is to identify basic human capabilities and incorporate them into a computational framework for selecting and executing actions. The second is to identify limitations and sources of performance variability associated with each capability and to constrain action selection mechanisms accordingly.

Psychological models have typically addressed the latter challenge at the expense of the former by employing simple action selection frameworks which raise relatively few questions about their psychological validity. The limitations of these frameworks arise less from their in-principle functionality than from their inexpressive notations for representing knowledge – i.e. knowledge about how to select and control action. For example, many models (e.g. EPIC [Kieras and Meyer, 1994]; SOAR [Laird, 1987]; ACT-R [Anderson, 1990]) use production systems for action selection. Production systems are convenient for modeling deductive reasoning, stimulus-response, and discrete action chaining, but are quite inconvenient for equally important capabilities such as multitask management, failure recovery , and continuous action control.

The simplicity of such production systems and similar frameworks aids in directing critical attention to variables of theoretical interest but also tends to limit application to highly constrained task environments. Since most human modeling research is evaluated using data from tasks and environments that are also highly constrained, improving the capabilities of these systems has not been a major concern.

Engineering models have employed powerful action selection frameworks such as scheduling algorithms (e.g. [Corker and Smith, 1993]) and hierarchical planners (e.g. [Card et al., 1983]), but neither these or any existing framework provides the diverse functional capabilities needed to operate in the most demanding task domains. This presents little problem for relatively simple environments such as that of the ATM for which the required cognitive, perceptual, and motor capabilities can easily be modeled. However, normal performance in other tasks such as operating an automobile or controlling air traffic require flexibility and sophistication that can not yet be duplicated by computer programs.

Fortunately, this situation is changing as more powerful and versatile approaches have been developed within the field of artificial intelligence. This thesis describes an attempt to adapt a particularly promising approach called "sketchy planning" [Firby, 1989] for use within a human model, to extend that approach in certain important ways, and to incorporate constraints and sources of performance variability needed to account for major classes of human error.

1.4.1 GOMS

GOMS, the most well-known and widely-used engineering model of human performance, provides a useful benchmark for comparing different modeling approaches. It also includes one of the most capable action-selection frameworks presently used for human modeling. Action choice derives from "how-to" knowledge represented as Goals, Operators, Methods, and Selection Rules, hence the GOMS acronym. **Goals** represent desired actions or world states --- e.g. *GOAL:(delete-file ?file)* to express a desire to remove (access to) a computer file identified by the variable *?file.* GOMS "interpreter" mechanisms determine behavior by mapping goal structures to sequences of **operators**, each representing a basic physical or cognitive skill such as shifting gaze to a new location, grasping an object, or retrieving a piece of information from memory.

The GOMS approach assumes that the modeled agent has already learned procedures for accomplishing any goal. These procedures, represented as **methods** in a "method library," are used to decompose a goal into subgoals and operators. For instance, the following method might be used to achieve the file deletion goal.

```
Method-126
  1. determine ?location of ?icon corresponding to ?file
  2. move-mouse-pointer to ?location
  3. press-with-hand mouse-button
  4. determine ?location2 of trash-icon
  5. move-mouse-pointer to ?location2
  6. release-hand-pressure mouse-button
```

In this case, the two steps of the method --- 3 and 6 --- correspond to operators and can thus be executed directly. The other steps create subgoals, each of which must be recursively decomposed using methods from the method library until an operator sequence is fully specified.

Selection rules are used to match a goal to a method. When alternative methods are available, the selection rule associated with a goal must decide between them. Selection rules represent a portion of the simulated person's knowledge about what constitutes appropriate action in a given situation. For instance, one alternative way to delete a file may be to use a search utility designed to locate named files in a large file directory structure, and then use the utility's delete option once the target file has been found. A selection rule for deciding between method-823 which encodes this procedure and method-126 above might incorporate knowledge that method-126 is more convenient and therefore preferable as long as the location of the file is known. Thus:

```
Selection-rule for goal:(delete-file ?file)
 IF (known (location ?file)) THEN do method-126 ELSE do method-823
```

Task analysis refers to the process of determining what the simulated operator should know about how to act in a given task domain and then representing this how-to knowledge using, in the case of GOMS, goals, operators, methods, and selection rules. The expressiveness of the GOMS language for describing how-to knowledge, together with the "smarts" built into the interpreter for this language, determine what kinds of human activities can be captured by a GOMS task analysis.

To evaluate how well the GOMS framework can be made to simulate human operator performance in the most demanding task environments, it is useful to consider what makes an environment demanding and what requirements these demands place on any system for deciding action. This problem of characterizing and matching agent capabilities to a given task environment, has long been a focus of research within subfields of artificial intelligence concerned with planning and plan execution. The resulting understanding can be roughly broken down into three areas: (1) capabilities needed to cope with uncertainty; (2) capabilities for managing limited resources; and (3) capabilities needed to manage multiple, interacting tasks.

1.4.2 Coping with uncertainty

Perhaps the most dramatic advances in AI approaches to action selection concern how agents can behave effectively in uncertain task environments. Early planning systems were designed to operate in very simple environments in which all relevant aspects of the current situation (world state) are known to the planner, no change occurs except by the planner's action, and all the planner's actions succeed all of the time. Most real-world domains are not so benign.

Uncertainty arises in many different ways and for a variety of reasons. First, many real task environments are far too *complex* to observe all the important events and understand all the important processes. Decisions must therefore sometimes be made on the basis of guesswork about what is currently true and about what will become true.

Similarly, real task environments are often *dynamic*. Agents and forces not under the planner's control change the world in ways and at times that the planner cannot reliably predict. Moreover, previously accurate knowledge of the current situation may become obsolete when changes occur without being observed. Finally, the real world often imposes variable amounts of *time-pressure* on an agent's ability to achieve its goals. This creates uncertainty about whether to start work on new goals, which active goals should be given priority, and which methods for achieving a goal should be selected when, as is often the case, methods vary on a speed-accuracy tradeoff.

Additional sources of uncertainty arise from the nature of the planning agent itself. Motor systems may be clumsy and imperfectly reliable at executing desired actions. Perceptual systems may distort their inputs and thus provide incorrect characterizations of observed events. Cognitive elements may lose or distort memories, fail to make needed inferences, and so on.

Together, these various sources of uncertainty have a profound effect in determining what kinds of capabilities an agent requires to perform effectively. For example, the possibility that some action will fail to achieve its desired effect means that an agent in the real (or realistic) world needs some way to cope with possible failure. Thus, it may need mechanisms to formulate explicit expectations about what observable effect its action should achieve, check those expectations against observed events, and then, if expectations fail, generate new goals to recover, learn and try again.

A further effect of uncertainty, possibly the most important in terms of its effect on planning research in the last decade, is that it is normally impractical to plan out future actions in detail. Early ("classical") planning systems took a set of goals as input and produced a fully specified sequence of actions to achieve those goals as output. For instance (to simplify slightly), a goal of driving home from work would be broken down into precisely timed, low-level actions such as applying a specified amount of torque to the steering wheel at a specified moment, or depressing the brake pedal a certain amount. Uncertainty about such things as the overall amount of traffic, road conditions, and the position and speed of individual vehicles, make it infeasible to plan ahead at this level.

Reactive planners (e.g. [Agre and Chapman, 1987]) abandon the idea of planning ahead entirely; instead, immediately executable actions are selected on the basis of current perceptions and stimulus-response rules. Although such systems do not hold internal state (memory) and do not make explicit inferences about hidden or future aspects of world-state, they have demonstrated surprisingly clever and sophisticated behavior. Most importantly, reactive planners avoid the necessity of deciding action while decision-relevant information is uncertain.

Of course, the reactive approach sacrifices many desirable attributes of more deliberative systems. For instance, lacking any ability to store and retrieve new memories, a reactive agent can not condition its behavior on its own past actions, and thus can not keep itself from engaging in futile or overly repetitious behavior. Nor can it reason about solutions to novel problems, maintain focused effort in a distracting environment, or interleave effort at multiple tasks.

Numerous frameworks combining reactive and deliberative approaches have been developed. In its original form, the GOMS framework was essentially a **hierarchical planner** [Friedland, 1979; Stefik, 1980], a kind of deliberative planner that provided an important, early foundation for certain deliberative-reactive hybrids. The idea of hierarchical planner is to work out the more abstract aspects of a plan before considering details. In making a plan to drive home from work, for example, one might first consider the overall route, then how to manage each leg along the route; low-level details such as braking and steering actions are considered last.

To enable a hierarchical planner to cope with uncertainty about future conditions requires two conceptually simple improvements: (1) the ability to delay consideration of the lower-level details of a plan until information about future conditions becomes available; and (2) the ability to execute fully detailed parts of the plan when other parts have not yet been worked out. These improvements add a crucial reactive capability. For instance, in controlling a simulated vehicle, such a system may decide to maintain a certain amount of pressure on the gas pedal at the current time without making any corresponding decision for even a few seconds into the future. If another vehicle unexpectedly moves across its path, the agent can shift from the gas pedal to the brake, and then back to the gas pedal. From the perspective of the planner, these actions are as much a part of the plan as any other, just as if the other vehicle's behavior had been anticipated and the braking and resumed acceleration had been decided much earlier.

This sort of augmented hierarchical planner is referred to as a *sketchy planner* [Firby, 1989], or less specifically as an *execution system*, reflecting the interleaving of planning with plan execution. A "highly interactive" version of GOMS developed by Vera and John [1992] uses what amounts to a simple version of this technology to play a video game; this dramatically improves on the uncertainty-handling capabilities of the original GOMS approach. Further gains, such as the ability to handle a continuously changing environment, enhanced failure recovery, and greater flexibility in adapting method choice to the current situation, could be achieved by adopting the full range of sketchy planing mechanisms described by [Firby, 1989].

Like most execution systems, all versions of GOMS employ a library of reusable, routine plans (methods) which further enable the system to cope with uncertainty. In particular, these plans are the result of an adaptive learning process (or an engineering effort meant to approximate adaptive learning) through which regularities in the task environment shape behavior. For instance, a commuter's routine plan for driving home from work is likely to reflect accumulated knowledge about which route is best. Moreover, the commuter may have several routine plans and a rule for deciding between them that incorporates further knowledge -- e.g.

that one route is better during "rush hour" when high traffic is likely, but another should be preferred all other times.

The ability of sketchy planners to delay action selection while waiting for additional information and its use of refined, reusable plans that incorporate adaptations to environmental regularities represent the two basic capabilities an agent must employ to cope with uncertainty. In particular, a capable agent must have the flexibility to control the timing (scheduling) of its action commitments and it must make use of whatever knowledge is available to reduce its uncertainty, even if that knowledge is heuristic, probablistic or incomplete.

1.4.3 Coping with limited resources

The GOMS language and interpreter together constitute an **action selection architecture**. Nothing about GOMS or any alternative action selection architecture limits its application to simulating human behavior. It could, for example, control a real robot, simulate a non-human animal, or simulate a group such as an ant colony, basketball team, or army. To add the necessary human dimension, modelers embed GOMS within a set of cognitive, perceptual, and motor elements – a **resource architecture** – such as the Model Human Processor (MHP) [Card et al., 1983].

Each resource in the MHP is associated with a set properties, including especially, limits on how quickly actions requiring that resource can be performed. For example, to execute a *press-button* operator, GOMS requires a hand resource for an amount of time determined by Fitts' Law [Fitts and Peterson, 1964]. Similarly, the act of selecting between methods using a selection rule requires an amount of time based on the number of alternative methods and other factors. The properties of agent resources partially determine the performance of the system. For instance, the time-requirements for actions imposed by resources determine the amount of time that will be required to carry out all steps of a method. The ability to predict method execution time can be of substantial practical benefit [Gray et al., 1993].

Neither GOMS-MHP or any other human model has captured the full range of resources and performance-limiting resource attributes. In addition to temporal characteristics, important resource attributes include: precision limits, capacity limits, fatigue characteristics, bias characteristics, and unique state. Limited **precision** refers to the tendency of a resource to perform more coarsely than may sometimes be desired. For instance, perceptual resources cannot discriminate perceptual features exactly. The visual system might be able to tell blue from red, but not from a similar shade of blue. A motion may be detected "off to the left," when more exact location information would be desirable. Cognitive and motor resources have analogous precision limits. For example, one may be able to move a grasped thread near the eye of a needle but still have trouble moving it through.

Capacity limits define value ranges over which resources can function and determine how performance degrades outside those ranges. Hand and arms have limited strength, can carry limited volume, and can only reach to a certain distance and in certain directions. Eyes can observe ahead, not behind, and only within the "visible" range of the electromagnetic spectrum. **Fatigue characteristics** define how function degrades as a resource is used, and how it recovers during rest. This includes such phenomena as eye-strain resulting from intense light, and loss of strength and coordination in continuously exercised large muscles.

Bias characteristics are systematic sources of inaccuracy or preference such as the tendency to believe visual information in the presence of conflicting information from other perceptual modalities, and right- or left-handedness. For practical applications, many of the most interesting kinds of bias are those affecting cognitive resources. For example, recency bias can cause people to repeat actions inappropriately. Frequency gambling bias [Reason, 1990] causes people to believe and act in ways that are normally (frequently) correct, sometimes even when counterevidence for the norm is perceptually available or easily retrieved from memory.

Unique state refers to the obvious but important fact that resources currently engaged in one activity or, more generally, existing in one state, cannot simultaneously be in another. The right hand cannot be on the steering wheel and grasping the gear shift at the same time. Eyes cannot look left while also looking right. Psychological research has begun to determine the identity and state parameters of cognitive resources. For example, memory retrieval processes appear to only be capable of processing one retrieval cue at a time [Carrier and Pashler, 1995], forcing partially serial execution on tasks that might seem executable in parallel.

From a practical engineering perspective, this rich variety of resource constraints represents something of a dilemma. On the one hand, modeling these constraints makes it possible to predict interesting aspects of human performance, some of which may be crucial in analyzing designs. On the other hand, such constraints may have wide-ranging implications for human behavior that complicate the process of producing a model. For example, consider the fact that human visual acuity is highest at the point of fixation and falls to nothing in the visual periphery. The straightforward consequence is that a human model should get more precise information about certain portions of the visual field than others in accordance with well-known findings about this fairly well-understood aspect of human performance.

The indirect consequences impose far greater demands on a useful model. In particular, humans largely circumvent the described limit on visual acuity by frequently shifting their gaze to new locations. A model that included the acuity limit but failed to model gaze shifting would be essentially blind. Gaze shifting, however, is quite complicated and not nearly so well understood as acuity. People shift gaze in a variety of patterns [Ellis and Stark, 1986] for a variety of purposes: to check out imprecisely specified events detected in the periphery of vision; to update potentially obsolete information about spatially separated objects; to sequentially search among moving and poorly defined targets (e.g. people in a crowd). Representing all of these activities entails a greater effort than that required to model the resource limit from which they arise. Thus, the process of constructing a capable but resource-limited human operator model should proceed by gradually adding limits and means of coping with these limits to a highly capable model architecture.

Generally speaking, three requirements must be met by a model able to cope with its own resource limitations. First, its action selection component must include operators able to manipulate or configure resources appropriately. For example, a model with acuity limits must include an operator for shifting gaze. Second, action selection mechanisms must be able to detect excessive demands on its resources. For example, when an information acquisition task makes excessive demands on working memory capacity, an agent might react by writing information down on a list. Similarly, when an object appears in the periphery of vision but its shape cannot be discriminated, the agent should be able to detect this and shift gaze for better resolution.

Third, coping with resource limitations requires a great deal of knowledge including knowledge of when, based on past experience, resource limits are likely to be exceeded, strategies for coping with or circumventing resource constraints, and knowledge about the world needed to carry out those strategies in diverse situations. All of this must be incorporated, implicitly or explicitly, into the representations of "how-to" knowledge that underly an agent's routine behavior. For example, an airline pilot is responsible for maintaining an up-to-date awareness of numerous instrument readings. Limits on visual processing make it impossible to read the instruments in parallel, so pilots are taught a procedure (scan pattern) for reading them in sequence.

Similarly, most people find it difficult to recall which items to buy at the grocery store and learn to circumvent their limited memory by writing needed items down on a list. Knowledge underlying the use of this strategy is incorporated into a variety of routine behaviors to insure that the list is updated when needs are newly discovered, brought to the store, and used once there. In real humans, knowledge for coping with limited resources is obtained by learning from experience or from others. For simulated humans operating with devices and in task environments that do not yet exist, learning is not a realistic option. Instead, modelers must employ knowledge engineering techniques such as those used in the construction of expert systems to identify requisite knowledge, and must then represent this knowledge in GOMS or some more expressive notation.

1.4.4 Managing multiple, periodic tasks

The third category of capabilities associated with action selection concerns the problem of managing multiple tasks. Early planners were designed with a number of assumptions that made multitask management unnecessary. For example, all goals were assumed to be known at the outset – i.e. no new goals would arise before the old goals had been accomplished – and thus, no ability to interrupt ongoing tasks for higher-priority new tasks was required. Of course, people interrupt and resume tasks all the time when carrying out everyday acivities such as driving, cooking, and answering telephone calls.

Early planners also assumed that the time-requirements for achieving goals were exactly known or else irrelevant, and that goal accomplishment was unconstrained by temporal factors such as deadlines, decay, and instability. Combined with the assumption that all action would succeed on the first try, this assumption allowed planners to schedule each planned action in advance. In the real world, uncertainty about how long tasks will take and when the desired result will be accomplished creates uncertainty about how tasks will interact; this fact has implications for every aspect of action selection and execution.

For instance, it implies that agents will often not know when an opportunity will arise to begin or resume action in pursuit of their goals. Thus, to take advantage of transient opportunities, action selection mechanisms must be able to shed, delay, interrupt, and resume task execution. And since alternative methods for achieving a goal may vary in time and required resources, the process of selecting a method must take overall workload and the specific resource requirements of other tasks into account. Numerous other action selection system requirements follow from the possibility that a task's control of a resource might be interrupted. For example, tasks such as driving a car cannot be interrupted abruptly without dangerous consequences; interruption must therefore be treated as a task in itself and handled by domain-specific procedures (e.g. pulling over to the side of the road when interrupting driving), not by a simple and uniform internal process in which control is given over to a new task.

The problem of multitask management is a very important one for human simulation since many of the task domains of greatest practical interest require extensive multitasking capabilities. Moreover, much of the performance variability that needs to be accounted for, including many of the errors people make, arise from cognitive processes underlying multitasking. For instance, when people forget to resume an interrupted task, significant, even catastrophic impact to the performance of the human-machine system may result.

The importance of multitasking capabilities for human modeling can be seen in the automatic teller machine example discussed in section 1.1. Earlier I suggested that any existing human simulation model could perform the ATM task. More specifically, any model could be made to perform the ATM task correctly. However, predicting that users would sometimes fail to retrieve their cards requires a model of the processes from which postcompletion errors arise. In particular, both common-sense and empirical evidence support the idea that postcompletion errors arise, in part, from competition for resources among separate tasks [Byrne and Bovair, 1997]. For example, a typical ATM user will have other things to do besides get money, and these tasks will compete for control of gaze, cognitive resources, and physical location. Combined with a tendency to reduce resource control priority for tasks whose main goal has been accomplished, competition from other tasks may result in premature loss of resource control by the uncompleted task.

GOMS provides negligible capability for controlling multiple tasks. The original GOMS framework [Card et al., 1983] and most variations (see [John and Kieras, 1994]) make a strong assumption of task independence, although the interactive version of GOMS described earlier [John and Vera, 1992] provides a limited ability to select between competing tasks. Psychological models such as EPIC have examined a primitive form of multitasking based on the dual-task experimental literature. Tasks in this model are simple and too brief to be meaningfully interrupted – e.g. pressing a button in response to a tone – though they can interact within the model's resource architecture. Among engineering models other than GOMS, OMAR [Deutsch, 1993] and MIDAS [Corker and Smith, 1993] incorporate limited multitask management functions such as the ability to interrupt tasks and to define their behavior while suspended.

Just as a capable agent must manage multiple, conceptually separate tasks, it must also be prepared to manage multiple instances (repetitions) of a single task. As with multitasking, the ability to manage repetitive, or *periodic*, behavior underlies normal performance at wide variety of tasks. In the everyday realm, tasks such as watering plants and checking the fuel gauge while driving require both world knowledge and general-purpose capabilities to insure that they are carried out often enough, but not too often.

Tasks such as watering plants impose direct costs if executed too often or too infrequently (the plants will die). Preventing overly repetitious behavior in this case requires mechanisms for storing and accessing memories of past behavior. Tasks such as checking the fuel gauge impose opportunity cost (to other tasks that need the gaze resource) if carried out more often than needed and risk (of running empty) if not carried out often enough. Action selection mechanisms must manage how such tasks compete for resources so that recency of execution affects task priority.

Repetition can arise as unneeded redundancy when similar goals are active. For example, a person may have several chores to do, all of which require traveling to a particular location. Managing repetition in this case means recognizing task redundancy and coordinating what would otherwise be considered independent tasks. Iterative repetition such as expelling air to blow up a balloon requires an ability to manage termination for innately repetitive tasks. Managing repetition that arises from task failure – e.g. repeatedly turning an ignition key to start a car – requires an ability to detect "futile loops" [Firby, 1989].

The ability to manage multiple, possibly periodic tasks is a relatively new development in AI planning systems. The problems involved have not been well-articulated (although see [Schneider and Detweiler, 1991]) and no well-known system addresses those problems in depth. As noted, deliberative or "classical" planning systems make simplifying assumptions that eliminate the need for such abilities; and in any case, such systems largely abandon the concept of separate tasks since all goals are planned for jointly. Reactive systems do not represent tasks and retain little of the knowledge or computational capacity needed to manage any aspect of behavior that takes place over more than a few moments of time.

Multitask and periodicity management issues have been explored almost exclusively within plan execution systems, especially including sketchy planners such as Firby's RAP execution system [Firby, 1989]. To incorporate the necessary abilities into a human operator model, sketchy planning methods need to be adapted and improved.

1.5 APEX

This dissertation describes an approach to modeling human behavior and a computer program called APEX that implements this approach. APEX is intended to advance on past modeling efforts in two ways. First, by incorporating a highly capable framework for selecting action, the model should be able to simulate human behavior in more diverse and more demanding task domains than previous models. Second, APEX is intended to provide useful predictions of human operator error.

Like the GOMS-MHP approach, APEX couples an *action selection architecture* with a *resource architecture*. Action selection refers to the process of controlling internal and external resources such as attention, voice and hands. A capable action selection system – i.e. one that can reliably achieve its goals in real or realistic environments – must be able to manage multiple, sometimes repetitive tasks using limited cognitive, perceptual, and motor resources in a complex, dynamic, time-pressured, and partially uncertain world. The APEX action selection component achieves, by this definition, a high degree of capability by combining an expressive

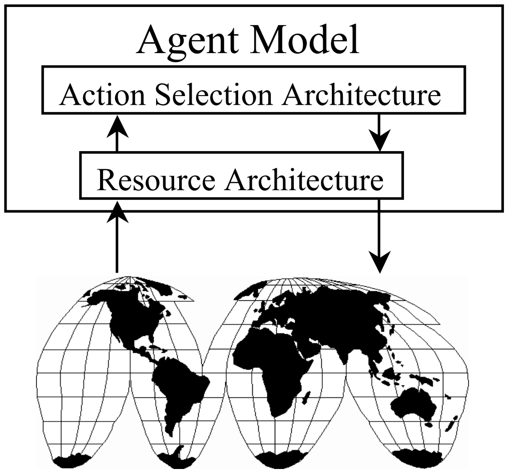


Figure 1.1 Agent model overview

procedure definition language for representing the simulated agent's how-to knowledge with *sketchy planning* [Firby, 1989] mechanisms able to execute procedures in demanding task environments.

Though APEX[1] is intended as a general-purpose design engineering tool, it has been developed and used almost exclusively in the domain of air traffic control (ATC). Air traffic control presents a variety of challenges for human modeling. In particular, ATC makes demands on aspects of human behavior such as multitask management and situation awareness that seem to be representative of the modeling challenges one would expect in other domains of practical interest such as air and ground vehicle piloting, and supervisory control of automated facilities. Moreover, increased ability to predict human performance at air traffic control tasks is likely to have near-term value since numerous improvements to existing ATC technologies are presently in development.

The use of ATC tasks to drive development has shaped the resulting model in at least one important way. Individual resource components in the APEX resource architecture have been articulated in greater or lesser detail as needed to account for performance variability in air traffic control scenarios. For example, the visual processing resource has been the object of much greater modeling effort then the kinesthetic processing resource. If the system had been developed to simulate, for example, aircraft piloting tasks, the kinesthetic sense would likely have been modeled in greater detail.

Human models have most often been employed to predict temporal properties of human performance, particularly the time needed to learn or to carry out a procedure [Landauer, 1991]. Other models focus on predicting design attributes such as text legibility and control reachability that arise from human perceptual or motor limitations. However, psychological models of *error* are rare [Byrne and Bovair, 1997] and engineering models even more so [Kitajima and Polson, 1995]. APEX incorporates a theory of skill-based errors derived largely from Reason's account of systematic slips and lapses [Reason, 1990].

As discussed in section 1.2 and 1.3, general knowledge about human performance becomes useful when applied to the analysis of specific and highly detailed scenarios. The amount of detail and diversity of scenarios that need to be considered increases dramatically as task environments become more complex, tasks become more prolonged, and the variety of

[1] APEX stands for Architecture for Procedure Execution. *Procedures* are the fundamental unit of how-to knowledge in APEX, combining the functions of GOMS methods and selection rules.

possible operating conditions increases. As with other design domains in which computer simulation has proven invaluable, domains such as air traffic control are far too complex and variable for unaided human designers to think about concretely. Consequently, ATC equipment and procedures must undergo frequent refinement to prevent problems that, had designers considered the circumstances in which they tend to occur, might well have been anticipated. The APEX approach is thus seen as a way of extending the discerning power of everyday psychology to more complex task environments than was previously possible.

1.6 Dissertation outline

This dissertation is divided into 8 chapters. **Chapter 1** summarized the state of the art in using simulations of human behavior to guide the design of human-machine systems, laid out a course for future developments, and introduced the APEX human operator model.

Chapter 2 discusses how to prepare an APEX model, employ it in simulation, and use the simulation results to guide design. An example air traffic control scenario used to illustrate this process illustrates the primary innovations of the model: the ability to simulate human behavior in a demanding task environment and the ability to predict an error that seems obvious from hindsight but might plausibly go undetected during design.

Chapters 3 covers the APEX action selection architecture, beginning with a description of sketchy planning mechanisms. The primary focus of the chapter is an overview of the procedure definition language (PDL) used to represent a simulated operator's how-to knowledge, followed by a discussion of several representation issues that arise when modeling agent behavior.

Chapter 4 describes extensions to the basic action-selection architecture and procedure definition language used to manage multiple tasks. The need for multitask management capabilities arise because tasks can interact, especially over the use of limited resources.

Chapter 5 describes the cognitive, perceptual, and motor model components which constitute the APEX resource architecture, and explains how a user can refine and extend this architecture as needed to improve the model's predictive capabilities.

Chapters 6 and **7** describes human error modeling in APEX with chapter 6 laying out the theoretical framework used to predict error and chapter 7 illustrating this approach with a number of simulated air traffic control scenarios.

Finally, **chapter 8** discusses what has been and what has yet to be accomplished in order to make computer simulation a practical tool for designing human-machine systems.

Chapter 2
Using APEX

2.1 Iterative design

The enormous cost of fielding complex human-machine system can be attributed in part to the cost of discovering and eliminating usability problems in its design. In general, evaluation costs increase as the design process progresses. By the time a system has come into use, fixing a design problem involves not only redesigning and retesting, but also modifying fielded devices and possibly retraining users. To manage engineering design costs, large new systems are usually developed by a process of *iterative design* [Gould, 1988; Shneiderman, 1992; Nielsen, 1993; Baecker, et al., 1995, p. 74]. As a design concept progresses from idea to fully fielded system, a process of evaluating design decisions follows each stage. If problems are discovered during evaluation, the system is partially redesigned and further evaluation takes place on the new version. This process is repeated until a satisfactory version results. Of course, the ability to determine whether the current version is satisfactory is limited by the effectiveness of the evaluation methods employed.

Evaluation methods applicable at a late design stage are generally more expensive but also more effective than methods that can be used at earlier stages. In particular, once a working prototype of the new system has been constructed, evaluation by user testing becomes possible. Observing users employing the system in a wide range of scenarios and operating conditions can tell a designer a great deal about how well it will function once in the field. This process is widely recommended in discussions of human factors and routinely practiced in the design of safety-critical and high-distribution systems.

Method	when	redesign cost	method use cost	demanding task environments	method effectiveness
User testing	late	high	high	yes	high
Guidelines	early	low	low	no	low
Walkthrough	early	low	low	no	medium
Simulation	early	low	medium	yes	medium

Figure 2.1 Comparison of usability evaluation methods

However, user testing suffers from a number of drawbacks and limitations. For instance, subjects are often more highly motivated than true end-users and, in some cases, become too knowledgeable about the developing system to be useful in discovering certain problems. Another drawback is cost. When designing new air traffic control systems, for example, such tests typically require hiring highly paid expert controllers as subjects, often for extended periods [Shafto and Remington, 1990; Remington and Shafto, 1990]. The limited amount of testing that results from high cost can stifle innovation, slow development, and even compromise safety.

Designers can reduce the amount of user testing required by discovering problems early in the design process, thus reducing the number of design iterations. To discover problems with usability, the primary early-phase evaluation method involves checking the design against human factors guidelines contained in numerous handbooks developed for that purpose (e.g. [Smith and Mosier; Boff et. al.]). Guidelines have proven useful for some design tasks (e.g. [Goodwin, 1983], but have a number of fairly well-known problems [Mosier and Smith, 1986; Davis and Swezey, 1983]. In particular, guidelines focus on static, relatively superficial factors affecting human-machine performance such as text legibility and color discrimination. But when addressing topics relating to the dynamic behavior of a system or to the mental activities of the user, guidelines are often lacking or are too general to be of much use. Thus, "for the foreseeable future, guidelines should be considered as a collection of suggestions, rather than distilled science or formal requirements. Understanding users, testing, and iterative design are indispensable, costly necessities. [Gould, 1988]."

Scenario-based approaches, including Cognitive Walkthrough [Polson, 1992], "thinking aloud," [Lewis, 1982], and human simulation modeling offer alternative methods for early-stage design evaluation. These techniques trade off some of the guideline-based method's generality for greater sensitivity to human cognitive factors and for an increased ability to predict performance in complex, dynamic task domains. These scenario-based approaches achieve some of the benefits of user testing at an early design stage using when no usable prototype has been constructed. Designers follow the behavior of a real or hypothetical user employing imaginary or simulated equipment to achieve specified task goals in specified operating conditions.

Focusing on specific scenarios allows designers to consider situation-dependent aspects of performance such as the varying relevance of different performance variables, the effects of changing workload, and the likelihood and consequences of interactions between a user's tasks. However, complexity and dynamic elements in a task domain pose difficulties for any scenario-based approach. While an improvement over guidelines in this respect, all of these approaches become more difficult to use in more demanding task domains as task duration, situation complexity, number of actors, number of activities that each actor must perform, and the number of scenarios that need to be considered all increase.

By exploiting the computer's speed and memory, human simulation modeling overcomes obstacles inherent in other scenario-based methods and thus has the greatest potential for predicting performance in more demanding task environments. A large, accurate memory overcomes the problem of tracking innumerable scenario events. Processing speed helps compensate for the need to examine more scenarios by, in principle, allowing each scenario to be carried out more quickly than in real-time. The computer's ability to function continuously adds further to the number of scenarios that may be explored.

However, despite its potential, human simulation has been used to inform design almost exclusively in simple design domains – i.e. domains where tasks are brief, situational complexity is low, few actors determine events, and so on. Predicting performance in more challenging task domains requires an operator model that can function effectively in demanding task environments. Existing human models have typically lacked several very important capabilities including those needed to cope with numerous kinds of uncertainty inherent in many task environments; manage limited cognitive, perceptual, and motor resources; and, manage multiple, periodic tasks. The APEX human operator model represents an attempt to incorporate such capabilities.

APEX has been developed in the domain of air traffic control (ATC), a task domain that presents a variety of challenges for human modeling – in particular, for coping with uncertainty, managing limited resources, and managing multiple tasks – that seem representative of the challenges one would expect in many other design domains of practical interest. Moreover, an increased ability to predict human performance at air traffic control tasks may have near-term value since numerous improvements to existing ATC technologies are presently in development.

1. Constructing a simulated world
2. Task analysis
3. Scenario development
4. Running the Simulation
5. Analyzing simulation results

Figure 2.2. Steps in the human simulation process

The next section describes a scenario that sometimes occurs in an APEX ATC simulation. The scenario illustrates how APEX simulation fits into the overall design process and exemplifies its use in predicting operator error. Subsequent sections discuss each of the five steps listed above for preparing and using an APEX model to aid in design.

2.2 Example scenario

At a TRACON air traffic control facility, one controller will often be assigned to the task of guiding planes through a region of airspace called an arrivals sector. This task involves taking planes from various sector entry points and getting them lined up at a safe distance from one another on landing approach to a particular airport. Some airports have two parallel runways. In such cases, the controller will form planes up into two lines.

Occasionally, a controller will be told that one of the two runways is closed and that all planes on approach to land must be directed to the remaining open runway. A controller's ability to direct planes exclusively to the open runway depends on remembering that the other runway is closed. How does the controller remember this important fact? Normally, the diversion of all inbound planes to the open runway produces an easily perceived reminder. In particular, the controller will detect only a single line of planes on approach to the airport, even though two lines (one to each runway) would normally be expected (see figure 2a and 2c).

However, problems may arise in conditions of low workload. With few planes around, there is no visually distinct line of planes to either runway. Thus, the usual situation in which both runways are available is perceptually indistinguishable from the case of a single closed runway (figure 2b and 2d). The lack of perceptual support would then force the controller to rely on memory alone, thus increasing the chance that the controller will accidentally direct a plane to the closed runway[2].

Designing to prevent such problems is not especially difficult – it is only necessary to depict the runway closure condition prominently on the controller's information display. The difficulty lies in anticipating the problem. By generating plausible scenarios, some containing operator error, APEX can direct an interface designer's attention to potential usability problems. Though perhaps obvious from hindsight, such errors could easily be overlooked until a late stage of design. The ability to explicate events (including cognitive events) leading to the error can help indicate alternative ways to refine an interface. For example, one of the difficulties in designing a radar display is balancing the need to present a large volume of information against the need to keep the display uncluttered. In this case, by showing how the error results from low traffic conditions, the model suggests a clever fix for the problem: prominently depict runway closures only in low workload conditions when the need for a reminder is greatest and doing so produces the least clutter.

[2] Examples of such incidents are documented in Aviation Safety Reporting System reports [Chappell, 1994] and in National Transportation Safety Board studies (e.g. [NTSB, 1986]).

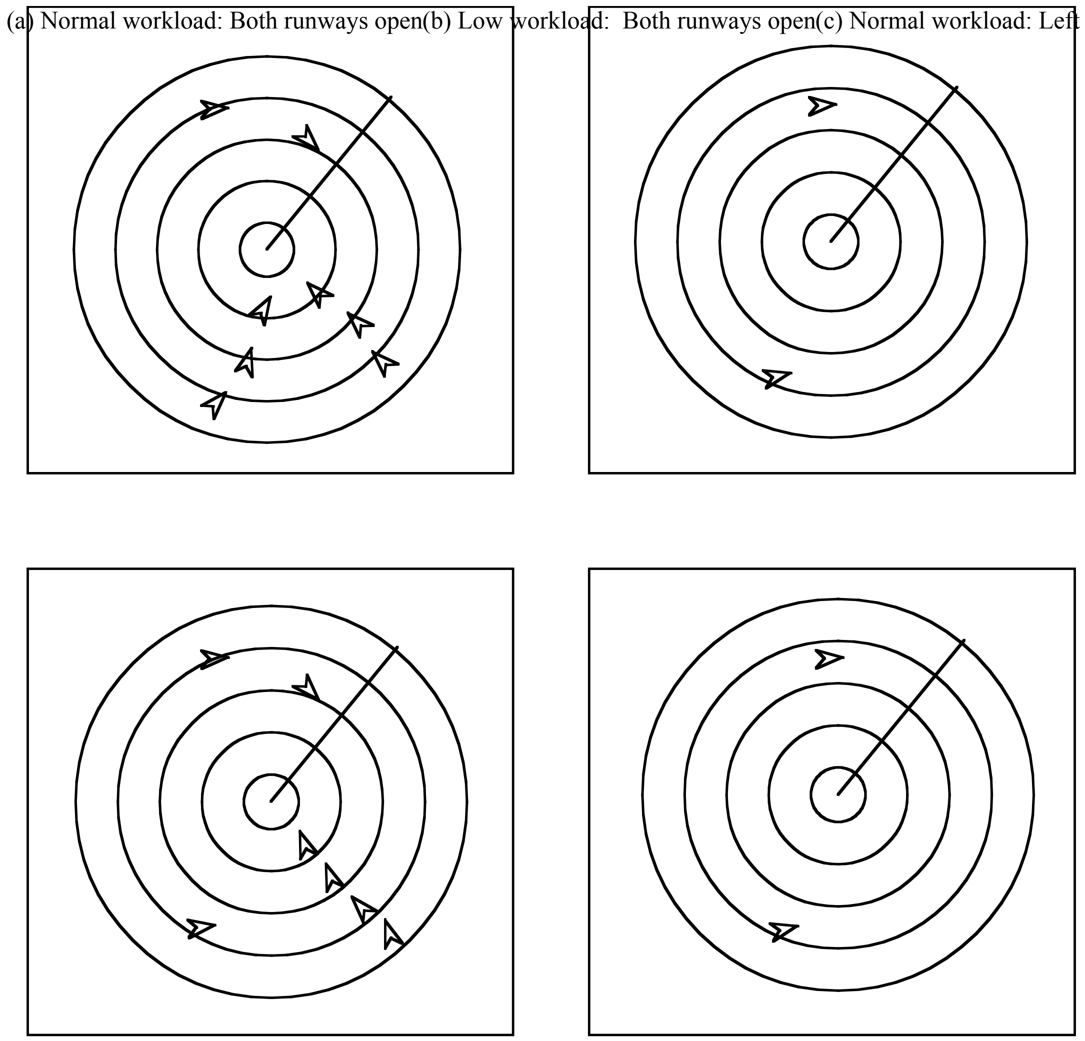

(a) Normal workload: Both runways open
(b) Low workload: Both runways open
(c) Normal workload: Left

Figure 2.3 Radar displays for approach control

2.3 Constructing a simulated world

The first step in simulating a human-machine system involves implementing software components specific to the task domain. Because the domain model used for simulation will almost inevitably require simplifying from the real domain, the exact nature of the tasks the simulated operators will have to carry out cannot be known until this step is accomplished. Constructing software to model the domain thus precedes representing task knowledge for the operator model. This software, the **simworld**, should include several components:

- A model of the **immediate task environment** including **equipment models** specifying the behavior of devices employed by the simulated operator. In ATC, these include a radar scope, two-way radio, and flightstrip board.
- A model of the **external environment** specifying objects and agents outside the operator's immediate task environment. In ATC, the external environment comprises a region of airspace over which the controller has responsibility, airspace outside that region's boundaries, a set of airplanes, and aircrews aboard those airplanes.
- A **scenario control** component that allows a user to define scenario events (e.g. airliner emergencies, runway closures) and scenario parameters (e.g. plane arrival rate) and then insures that these specifications are met in simulation. See section 2.5.

In addition, a **simulation engine** controls the passage of simulated time and mediates interactions within and among all simworld and simulated operator components. A simulation engine provided by the CSS simulation environment, discussed in section 2.6, is currently used to run the APEX human operator model as well as the air traffic control simworld described below.

2.3.1 Air traffic control – a brief overview

APEX has been specified to carry out controller tasks at a simulated terminal radar control (**TRACON**) facility. Controllers at a TRACON manage most of the air traffic within about 30 miles of a major airport. This region is situated within a much larger airspace controlled by an air route traffic control center (ARTCC) – usually just called "Center." TRACON space encompasses small regions of "Tower" airspace, each controlled by a major or satellite airport within the TRACON region. Airspace within a TRACON is normally divided into sectors, each managed by separate controllers. Pilots must obtain controller permission to move from one sector or airspace regime to another.

Controllers and pilots communicate using a two-way radio, with all pilots in a given airspace sector using the same radio frequency. Since only one speaker (controller or pilot) can broadcast over this frequency at a time, messages are kept brief to help control "frequency congestion." Controllers manage events in their airspace primarily by giving **clearances** (authorizations) to pilots over the radio. The most common clearances are:

- **handoffs**: clearances that permit a plane to enter one's airspace or, conversely, that tell a pilot about to exit one's airspace to seek permission from the next controller
- **altitude clearances**: authorizations to descend or climb. Used at a TRACON mostly to manage takeoffs and landings, but also to maintain safe separation between planes.
- **vectors**: i.e. clearances to change heading. The new heading may be specified as an absolute compass direction (e.g. "two seven zero" for East), as a turn relative to the current heading (e.g. "ten degrees left"), or with respect to a named geographical position appearing on navigational charts called a **fix** (e.g. "go direct to DOWNE").
- **speed clearances**: authorizations to change airspeed. Managing airspeeds is the most difficult, but usually the best, way maintain aircraft separation and to space arriving planes for landing.

Clearances are issued according to a standard phraseology [Mills and Archibald, 1992] to minimize confusion. For example, to clear United Airlines flight 219 for descent to an altitude of 1900 feet, a controller would say, "United two one niner, descend and maintain one thousand nine hundred." The pilot would then respond with a **readback** – "United two one niner, descending to one thousand nine hundred" – thus confirming to the controller that the clearance was received and heard correctly.

The **radar display** is the controller's main source of information about current airspace conditions. Each aircraft is represented as an icon whose position on the display corresponds to its location above the Earth's surface. Planes equipped with a device called a C- or S-mode transponder, including all commercial airliners, cause an alphanumeric **datablock** to be displayed adjacent to the plane icon. Datablocks provide important additional information including altitude, airspeed, airplane type (e.g. 747), and identifying **callsign**. Further information, especially including the airplane's planned destination, can be found on paper **flightstrips** located on a "flightstrip board" near the radar display.

As a plane approaches TRACON airspace from a Center region, it appears on the scope as a blinking icon. The controller gives permission for the plane to enter – i.e. accepts a handoff – by positioning a pointer over the icon and then clicking a button. The two-way radio on board the aircraft automatically changes frequency, allowing the pilot to communicate with the new controller. Some planes are not equipped for automatic handoffs, in which case a specific verbal protocol is used:

> *Example*: as a small Cherokee aircraft with callsign 8458R approaches Los Angeles TRACON airspace, the pilot manually changes the radio setting and announces, "LA approach, Cherokee eight four five eight romeo, ten miles north of Pasadena, at four thousand feet, landing." After detecting the plane on the radar scope, the controller announces "Cherokee eight four five romeo, radar contact," thereby clearing the plane to operate in LA TRACON airspace.

Standard operating procedures specify nearly every aspect of routine air traffic control at a TRACON, including the time window within which certain clearances should be issued and the flightpaths planes should be made to traverse on departure from and landing approach to airports. To continue with the previous example, the following event sequence illustrates a typical (though simplified) landing approach:

- After announcing radar contact, the controller locates the Cherokee's paper flight strip, determines that its destination is Los Angeles International airport (LAX), and selects an appropriate path from the plane's present position.
- The controller vectors the plane along the first leg on this path, saying "Cherokee five eight romeo, cleared direct for DOWNE." The pilot acknowledges with a readback.
- While the plane travels to the DOWNE fix, the controller observes it periodically to insure separation from other aircraft and to determine a safe time to clear it to the correct altitude for an LAX final approach. When appropriate, the controller says, "Cherokee five eight romeo, descend and maintain one thousand nine hundred."
- As the Cherokee approaches DOWNE, the controller selects a preferred runway and then locates a gap in the line of planes approaching that runway. Vectors and speed clearances are used to maneuver it into the gap at safe distance from other aircraft. For example, the plane may need to be 5 miles behind a 747 and 3 miles ahead of whatever follows.
- Finally, as the plane nears LAX Tower airspace, the controller initiates a handoff to Tower by saying "Cherokee five eight romeo, cleared for ILS approach. Contact tower at final approach fix."

2.3.2 The ATC simulator: defining an airspace

A TRACON is typically divided into separate airspace sectors, each handled by one or more individual controllers. The number of sectors handled by a controller usually varies over the course of a day to reflect the amount of expected air traffic, thereby managing the number of planes any particular controller needs to handle. For simplicity, the ATC simulation software, **ATCworld**, divides the overall airspace into an **arrivals sector** and a **departures sector**, each handled by a single controller. Examples throughout this document will center on the arrival sector controller at Los Angeles TRACON.

Users can easily define new airspace models in ATCworld. Such models consist of three kinds of objects: airports, fixes, and regions. Defining an airport or fix causes all simulated pilots in ATCworld to know its location; the controller can thus vector planes "direct to" that location. Defining an airport also creates an ATC Tower to which control of a planes can be handed off. When control of a plane passes to an airport Tower, the plane icon on the simulated radar display disappears soon thereafter.

Regions define operationally significant areas of airspace, possibly but not necessarily corresponding to legal divisions, and not usually encompassed by explicit boundaries on the display. They provide a usefully coarse way to represent plane location, allowing a controller to refer to the area, e.g., "between DOWNE and LAX." The ability to consider airspace regions allows the simulated controller to assess air traffic conditions, facilitates detection of potential separation conflicts, and provides a basis for determining when planes have strayed from the standard flightpath. Regions are essentially psychological constructs and are therefore properly part of the agent model, not the domain model. However, regions need to be represented in the same coordinate system as fixes and airports, making it convenient to specify all of them together.

2.3.3 The ATC simulator: controller tasks

In ATCworld, as in the real world, the task of handling an arrival is entirely routine. Most planes arrive from Center space via one of a few pre-established airspace "corridors." The controller periodically checks for new arrivals, represented as blinking plane icons, and then accepts control from center by clicking a mouse button over their icons. Once control of a new plane has been established, the paper flight strip associated with the plane is consulted to determine the flight's planned destination and then marked (or moved) to indicate a change from pending to active status. ATCworld uses “electronic flightstrips” in accordance with somewhat controvercial proposals to transfer flightstrip information to the controller’s information display [Stein, 1993; Vortac et al., 1993].

The task of routing a plane to its destination – either an airport or a TRACON airspace exit point – proceeds in ATCworld the same as it does in reality (see example in previous section), but with several simplifying assumptions. These include:

- Controllers do not need to communicate with one another; in reality controllers sometimes need to ask about pending handoffs or to request changes in traffic volume
- Controllers start with no aircraft in their airspace; real controllers must take over an active airspace from a previous controller
- The display pointer is controlled by a mouse; in reality, a trackball is used
- Aircraft include only commercial and private airplanes, not helicopters, military jets, balloons, gliders, etc..
- The simulated controller only gives clearances, no advisories
- In the current model, there is never any weather – i.e. no wind or precipitation

While controllers’ tasks are mostly simple and routine when considered in isolation, the need to manage multiple tasks presents significant challenge. For instance, the controller cannot focus on one aircraft for its entire passage through TRACON airspace, but must instead interleave effort to handle multiple planes. Similarly, routine scanning of the radar display to maintain awareness of current conditions often must be interrupted to deal with situations discovered during the scanning process, and then later resumed. A further source of challenge is the possibility that certain unusual events may arise and require the controller to adapt routine behavior. For example, if one of the runways at LAX closes unexpectedly, the controller will have to remember to route planes only to the remaining open runway and may have to reduce traffic flow in certain regions to prevent dangerous crowding.

2.4 Task analysis

APEX, like most other human simulation models, consists of general-purpose components such as eyes, hands, and working memory; it requires the addition of domain-specific knowledge structures to function in any particular task domain. **Task analysis** is the process of identifying and encoding the necessary knowledge [Mentemerlo and Eddowes, 1978; Kirwan and Ainsworth, 1992]. For highly routinized task domains such as air traffic control, much of the task analysis can be accomplished easily and fairly uncontroversially by reference to published procedures.

For instance, to clear an airplane for descent to a given altitude, a controller uses a specific verbal procedure prescribed in the controller phraseology handbook (see [Mills and Archibald, 1990]) – e.g. "United two one niner, descend and maintain flight level nine thousand." Other behaviors, such as maintaining an awareness of current airspace conditions, do not correspond to any written procedures. These aspects of task analysis require inferring task representation from domain attributes and general assumptions about adaptive human learning processes.

This section introduces the notational formalism (PDL) used in APEX to represent task analyses and discusses the role of adaptive learning in determining how agents come to perform tasks. Detailed discussion of the task analysis used for air traffic control simulation appears in chapters 3 and 6.

2.4.1 An expressive language for task analyses

In APEX, tasks analyses are represented using the APEX Procedure Definition Language (PDL), the primary element of which is the **procedure**. A procedure in PDL represents an operator's knowledge about how to perform routine tasks. For instance, a procedure for clearing a plane to descend has the following form:

```
(procedure
   (index   (clear-to-descend ?plane ?altitude))
   (step s1 (determine-callsign-for-plane ?plane => ?callsign))
   (step s2 (say ?callsign) (waitfor ?s1))
   (step s3 (say "descend and maintain flight level") (waitfor ?s2))
   (step s4 (say ?altitude) (waitfor ?s3))
   (step s5 (terminate) (waitfor ?s4)))
```

The **index clause** in the procedure above indicates that the procedure should be retrieved from memory whenever a goal to clear a given plane for descent to a particular altitude becomes active. **Step clauses** prescribe activities that need to be performed to accomplish this. The first step activates a new goal: to determine the identifying callsign for the specified airplane and to make this information available to other steps in the procedure by associating it with the variable ?*callsign*. Achieving this step entails finding a procedure whose index clause matches the form

```
(determine-callsign-for-plane ?plane)
```

and then executing its steps. After this, *say* actions prescribed in steps s2, s3, and s4 are carried out in order. This completes the phrase needed to clear a descent. Finally, step s5 is executed, terminating the procedure.

Although PDL will be described in detail in the next chapter, a few comments about the above procedure are relevant to the present discussion. First, the activities defined by steps of a PDL procedure are assumed to be concurrently executable. When a particular order is desired, this must be specified explicitly using the **waitfor clause**. In this case, all steps but the first are defined to wait until some other task has terminated. Second, although this task is complete when all of its steps are complete, it is sometimes desirable to allow procedures to specify more complex, variable completion conditions. For example, it may be useful to allow race conditions in which the procedure completes when any of several steps are complete. Thus, rather than handle termination uniformly for all procedures, termination conditions must be notated explicitly in each procedure.

The ability to specify how concurrent execution should be managed and to specialize termination conditions for each procedure exemplify an attempt with PDL to provide a uniquely flexible and expressive language for task analysis. In particular, PDL can be considered an extension to the GOMS approach in which tasks are analyzed in terms of four constructs: goals, operators, methods, and selection rules. Procedure structures in PDL combine and extend the functionality provided by GOMS methods and selection rules. GOMS operators represent basic skills such as pressing a button, saying a phrase, or retrieving information from working memory; executing an operator produces action directly. PDL does not produce action directly, but instead sends action requests (signals) to cognitive, perceptual, and motor resources in the APEX resource architecture. What action, if any, results is determined by the relevant resource component.

It is important to distinguish PDL procedures from externally represented procedures such as those that appear in manuals. PDL procedures are internal representations of how to accomplish a task. In some cases, as above, there is a one to one correspondence between the external prescription for accomplishing a task and how it is represented internally. But written procedures might also correspond to multiple PDL procedures, especially when written procedures cover conditional activities (i.e. carried out sometimes but not always) or activities that take place over a long period of time. Similarly, PDL procedures may describe behaviors such as how to scan the radar display that result from adaptive learning processes and are never explicitly taught.

2.4.2 Approximating the outcome of adaptive learning

Task analysis is often used to help designers better understand how human operators function in *existing* human-machine systems (e.g. [Hutchins,1995]). In such cases, task analysis can be usefully (though not altogether accurately) viewed as a linear process in which a task analyst observes operators performing their job, infers underlying cognitive activities based on regularities in overt behavior, and then represents these activities using a formal notation appropriately defined by some general cognitive model.

A different process is required to predict how tasks will be carried out with newly designed equipment and procedures. In particular, analysis can no longer start with observations of overt behavior since no real operators have been trained with the new procedures and no physical realization of the new equipment exists. Instead, cognitive structures underlying behavior must be inferred based on task requirements and an understanding of the forces that shape task-specific cognition: human limitations, adaptive learning processes, and regularities in the task domain.

For example, to model how a controller might visually scan the radar display to maintain awareness of current airspace conditions, an analyst should consider a number of factors. First, human visual processing can only attend to, and thus get information about, a limited portion of the visual field at any one time. By attending to one region of the display, a controller obtains an approximate count of the number of planes in that region, identifies significant plane clusters or other Gestalt groups, and can detect planes that differ from all others in the region on some simple visual property such as color or orientation.

But to ascertain other important information requires a narrower attentional focus. For example, to determine that two planes are converging requires attending exclusively to those planes. Similarly, to determine that a plane is nearing a position from which it should be rerouted requires attending to the plane or to the position. These visual processing constraints have important implications for how visual scanning should be modeled. For example, to maintain adequate situation awareness, the model should shift attention not only to display regions but also to individual planes within those regions.

An assumption that the human operator adapts to regularities in the task environment has further implications. For instance, if a certain region contains no routing points and all planes in the region normally travel in a single direction, there would usually be no reason to attend to any particular plane unless it strayed from the standard course. Adaptive mechanisms could modify routine scanning procedures to take advantage of this by eliminating unnecessary attention shifts to planes in that region. This saves the visual attention resource for uses more likely to yield important information.

A fully mature approach to human modeling will require techniques for identifying or predicting regularities in the domain and detailed guidelines for predicting how adaptive learning processes will shape behavior in accordance with these regularities. A few such guidelines will be considered in discussions of particular knowledge representation problems in chapter 3 (qv. [Freed and Remington, 1997]), and somewhat more general principles will be discussed in chapter 8 (qv. [Freed and Shafto, 1997]). However, the present work does not come close to resolving this important issue.

2.5 Scenario development

The third step in preparing an APEX simulation run is to develop scenarios. A scenario specification includes any parameters and conditions required by the simulated world. In general, these can include initial state, domain-specific rate and probability parameters, and specific events to occur over the course of a simulation. In the current implementation of ATCworld, initial conditions do not vary. In particular, the simulated controller always begins the task with an empty airspace (rather than having to take over an active airspace) and with the same set of goals: maintain safe separation between all planes, get planes to their destination in a timely fashion, stay aware of current airspace conditions, and so on.

At minimum, an ATCworld scenario must include a duration D and an aircraft count C. The scenario control component will randomly generate C plane arrival events over the interval D, with aircraft attributes such as destination, aircraft type, and point of arrival determined according to default probabilities. For instance, the default specifies that a plane's destination will be LAX with p(.7) and Santa Monica airport with p(.3). The default includes conditional probabilities – e.g. the destination airport affects the determination of airplane type – e.g. a small aircraft such as a Cherokee is much more likely to have Santa Monica as its destination.

Scenario definitions can alter these default probabilities, and thereby affect the timing and attributes of events generated by the scenario control component. Users can also specify particular events to occur at particular times. For example, one might want to specify a number of small aircraft arrivals all around the same time in order to check how performance is affected by a sudden increase in workload. Currently, arrivals are the only kind of event that the scenario control component generates randomly. Special events such as runway closures and aircraft equipment failures must be specified individually.

2.6 Running the Simulation

To employ the operator model, world model, and scenario control elements in simulation requires a simulation engine. APEX currently uses the simulation engine provided by CSS [Remington, et al., 1990], a simulation package developed at NASA Ames that also includes a model development environment, a graphical interface for observing an ongoing simulation, and mechanisms for analyzing and graphing temporal data from a simulation run.

CSS simulation models consist of a network of **process** and **store** modules, each depicted as a "box" on the graphical interface. Stores are simply repositories for information, though they may be used to represent complex systems such as human working memory. Process modules, as the name implies, cause inputs to be processed and outputs to be produced. A process has five attributes: (1) a name; (2) a body of LISP code that defines how inputs are mapped to outputs; (3) a set of stores from which it takes input; (4) a set of stores to which it provides output; (4) and a stochastic function that determines its finishing time – i.e. how much simulated time is required for new inputs to be processed and the result returned as output. A process is idle until a state change occurs in any of its input stores. This activates the process, causing it to produce output in accordance with its embedded code after an interval determined by its characteristic finishing time function.

The CSS simulation engine is an event-driven simulator. Unlike time-slice simulators which advance simulated time by a fixed increment, an event-driven simulator advances until the next active process is scheduled to produce an output. This more efficient method makes it practical to model systems whose components need to be simulated at very different levels of temporal granularity. In particular, the APEX human operator model contains perceptual processes that occur over tens of milliseconds, motor and cognitive processes that take hundreds of milliseconds and (links to) external simworld processes modeled at the relatively coarse temporal granularity of seconds. CSS provides further flexibility by allowing processes to run concurrently unless constrained to run in sequence.

The process of incorporating a model into a CSS framework is fairly straightforward, but a user must decide how much detail to include regarding the model's temporal characteristics. In the simplest case, one could model the world and the operator each as single processes. Because processes can have only a single finishing time distribution, such a model would assume the same distribution for all operator activities. For instance, a speech act, a gaze shift, a grasp action, and a retrieval from memory would all require the same interval.[3] The process-store network used to simulate and visualize APEX behavior models each component of the APEX resource architecture as a separate process (see section 1.4.3 and chapter 5 for a description of the resource architecture).

Once the process-store network has been constructed, and simworld and APEX elements incorporated into the code underlying processes, the simulation can be run. CSS provides a "control panel" window with several buttons. SHOW toggles the visualization function, causing information in processes and stores to be displayed and dynamically updated. START and STOP initiate and freeze a simulation run. STEP causes time to advance to the next scheduled process finish event, runs the scheduled process, and then freezes the simulation.

2.7 Simulation analysis

The final step in using APEX is to analyze the results of simulation. CSS provides tools for analyzing and graphing temporal aspects of behavior. For example, if interested in predicting how much time the controller took to respond when a new airplane appeared on the radar display, the modeler could specify that interest when constructing the process-store network (see [Remington et al., 1990] for how this is accomplished). CSS automatically stores specified timing values from multiple simulation runs and graphs the data on demand.

[3] Actions could be defined to take place over multiple activations of the process. For example, assume that the human operator process (HP) has a characteristic mean finishing time of 50ms and that speech is to be modelled so that one word is produced every 200ms. In that case, the speech resource model would keep track of HP activations, causing one word to be emitted every fourth call. Alternately, speech could be associated with its own process with a mean finishing time of 200ms. The advantage of decomposing the model into multiple processes is that there is no need to generate or keep track of process activations that do not directly produce events. Moreover, coarse process decomposition forces to model activities at finer levels of temporal granularity and thus undermines the advantage of event-driven simulation.

2.7.1 Design-facilitated errors

APEX is intended to help predict **design-facilitated errors** – i.e. operator errors that could be prevented or minimized by modifying equipment or procedure design. The current approach, detailed in chapter 6, assumes that people develop predictable strategies for circumventing their innate limitations and that these strategies make people prone to error in certain predictable circumstances. For instance, to compensate for limited memory, people sometimes learn to rely on features of their task environment to act as a kind of externalized memory. If, for whatever reason, the relied-on feature is absent when it should be present (or vice-versa), error may result.

In the *wrong runway scenario* described in section 2.1, the controller's error stemmed from reliance on a visually observable feature – an imbalance in the number of planes approaching to each runway – to act as a reminder of runway closure. When planeload dropped too low for this feature to remain observable, the controller reverted to a behavior consistent with its absence. In particular, the controller selected a runway based on factors that had nothing to do with runway availability, such as airplane type and relative proximity to each runway approach path.

When this error occurs in simulation, the sequence of events that led up to it can be extracted from the **simulation trace**, a record of all the events that occurred during the simulation run. However, this "raw" event data is not very useful to a designer. To inform the design process, the events must be interpreted in light of general knowledge about human performance. For instance, most errors can be partially attributed to the non-occurrence of normal events. The raw simulation data will not contain any reference to these events, so normative knowledge must be used to complete the causal story that explains the error.

As an additional constraint on what constitutes a useful analysis, the explanation for an error must assign blame to something that the designer has control over ([Owens, 1990] makes a similar point). For instance, citing human memory limitations as a cause of the above described error is correct, but not very useful. In contrast, blaming the failure on the absence of expected perceptual support for memory implies ways of fixing the problem. The designer could enforce the perceptual support (in this instance, by insuring that planeload never drops too low), provide alternative perceptual support (a runway closure indicator on the display), or train the operator not to expect perceptual support and to take other measures to support memory.

2.7.2 Error patterns

One way to facilitate the generation of useful simulation analyses – analyses in which, e.g., non-occurring normal events are made explicit and causal explanations trace back to elements of the task environment that the designer might be able to control – is to represent general knowledge about the cause of error in **error patterns**. An error pattern is a specific type of explanation pattern [Schank, 1986] – i.e. a stereotypical sequence of events that end in some kind of anomaly that needs to be explained (an error in this case). When an error occurs in simulation, error patterns whose characteristic anomaly type match the "observed" error are compared against events in the simulation trace. If the pattern matches events in the simulation trace, the pattern is considered an explanation of the error.

Error patterns derive from a general theory of what causes error. Chapter 6 describes the theoretical approach used to simulate and explain the wrong runway error. To make the idea of error patterns concrete, the following example (next page) describes a simpler form of error.

Because an APEX human operator model can only approximate a real human operator, error predictions emerging from simulation will not necessarily be problems in reality. The designer must evaluate the plausibility and seriousness of any error predictions on the basis of domain knowledge and a common sense understanding of human behavior. As noted in section 1.2, current scientific knowledge about human error-making is inadequate for making detailed predictions. APEX attempts to make designers more effective at applying their common sense knowledge about when and why people make errors. Thus, the requirement that designers evaluate the plausibility of model predictions should be considered compatible with the APEX approach.

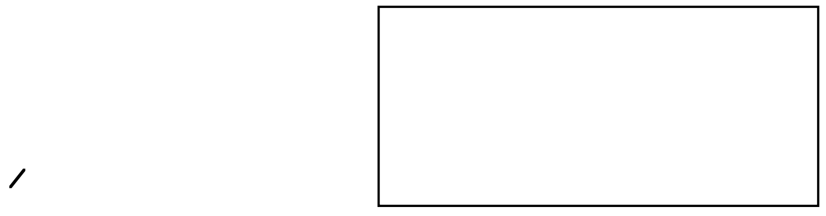
One other aspect of simulation analysis presents more of a problem. Currently, the modeler must interpret simulation event data "by hand" on the basis of informally specified error pattern knowledge. This approach is far from ideal and, given the massive mount of simulation data that must be examined, probably unacceptable for practical use. To automate analysis, simulation mechanisms must be augmented to check observed (i.e. simulated) behavior against expected behavior and to signal errors when specified deviations occur. Error patterns indexed by the anomaly and successfully matched against the simulation trace would then be output to the user as error predictions.

Example:

The UNIX command **rm** normally deletes files, thus freeing up hard drive space, but can be redefined to move files into a "trash" directory instead. A user unaware that rm has been redefined may try to use it to make space on the disk for a new file. After typing the command, the system returns a new command line prompt but does not describe the outcome of the command just given. As this response was the expected result of a successful delete, the user believes that the action succeeded in freeing up disk space.

The following general causal sequence constitutes an error pattern that can be specified to match the above incident:

agent wants G, achievableby action A with prereq Pagent believes not(P)agent believes action Bresults in P



Chapter 3
The Action Selection Architecture

As described in section 1.5, the APEX human operator model combines two components: an **action selection architecture** (ASA) that controls the simulated operator's behavior, and a **resource architecture** (RA) that constrains action selection mechanisms to operate within human perceptual, cognitive, and motor limitations. This chapter describes the basic elements of the ASA.

Action selection refers to the process of carrying out tasks by controlling **resources** such as attention, voice and hands. To model human operators in air traffic control and many other domains of practical interest, an action selection system must be able to act *capably* in realistic task environments – i.e. it must be able to manage multiple, sometimes repetitive tasks using limited cognitive, perceptual, and motor resources in a complex, dynamic, time-pressured, and partially uncertain world. The APEX action selection component achieves, by this definition, a high degree of capability by combining an expressive **procedure definition language** for representing the simulated agent's how-to knowledge with **sketchy planning** mechanisms based on those described in [Firby, 1989].

3.1 The Sketchy Planner

The idea of a sketchy plan is that it is useful to specify future actions in an abstract "sketchy" way and then gradually fill in the details as information becomes available. For example, a fully detailed plan for driving home from work would consist of precisely timed, low-level actions such as applying a specified amount of torque to the steering wheel at a specified moment, depressing the brake pedal a certain amount, and so on. Though it is clearly unfeasible to plan to this level of detail, an agent may find it useful to decide which route to take home at the time of departure. The route can be though of as an abstract description of the fully detailed plan.

As the plan is executed and new information becomes available, steps of this plan can be broken down into less abstract steps and eventually into low-level actions. For example, information about traffic conditions on a given road can help the agent decide which lane it should currently travel on. Information about the observed motion of nearby vehicles can be used to decide whether to brake or accelerate at a given moment. Even though many such actions will be decided reactively, they fit directly into a hierarchy of tasks leading from the initial task (returning home) through increasingly specific subtasks, down to the lowest level actions.

3.1.1 Procedures and tasks

In many domains, driving and air traffic control for example, most activities can be seen as combinations and simple variations of routine behavior. The routineness of these activities makes it possible to model them in terms of generic and highly reusable procedural knowledge structures. Domain expertise is often thought to be founded on a sufficient repertoire of such structures [DeGroot, 1978; Card et al., 1983; Newell, 1990]. In the APEX sketchy planner, routine procedural knowledge is represented using **procedure** structures which are stored in the planner's **procedure library** (see figure 3.1) and defined using the APEX Procedure Definition Language (PDL). For example, consider the PDL procedure discussed in section 2.3 which represents air traffic control actions used to authorize an aircraft descent.

```
(procedure
    (index   (clear-to-descend ?plane ?altitude))
    (step s1 (determine-callsign-for-plane ?plane => ?callsign))
    (step s2 (say ?callsign) (waitfor ?s1)
    (step s3 (say descend and maintain flight level) (waitfor ?s2))
    (step s4 (say ?altitude) (waitfor ?s3))
    (step s5 (terminate) (waitfor ?s4)))
```

The procedure is retrieved and executed whenever an enabled **task** can be matched to the procedure's **index clause**. For example, when a task of the form

{clear-to-descend plane-17 3500}

becomes enabled, the above procedure is retrieved and the variables *?plane* and *?altitude* in the procedure's index clause are assigned the values *plane-17* and *3500* respectively. Task structures, representations of the agent's intended behavior, are stored on the planner's **agenda** and can exist in any of several states. Pending tasks are waiting for some precondition or set of preconditions to become satisfied. Once this occurs, the task becomes **enabled**. Enablement signals the planner that the task is ready to be carried out (executed). Once execution has been initiated, the task's state changes to **ongoing**.

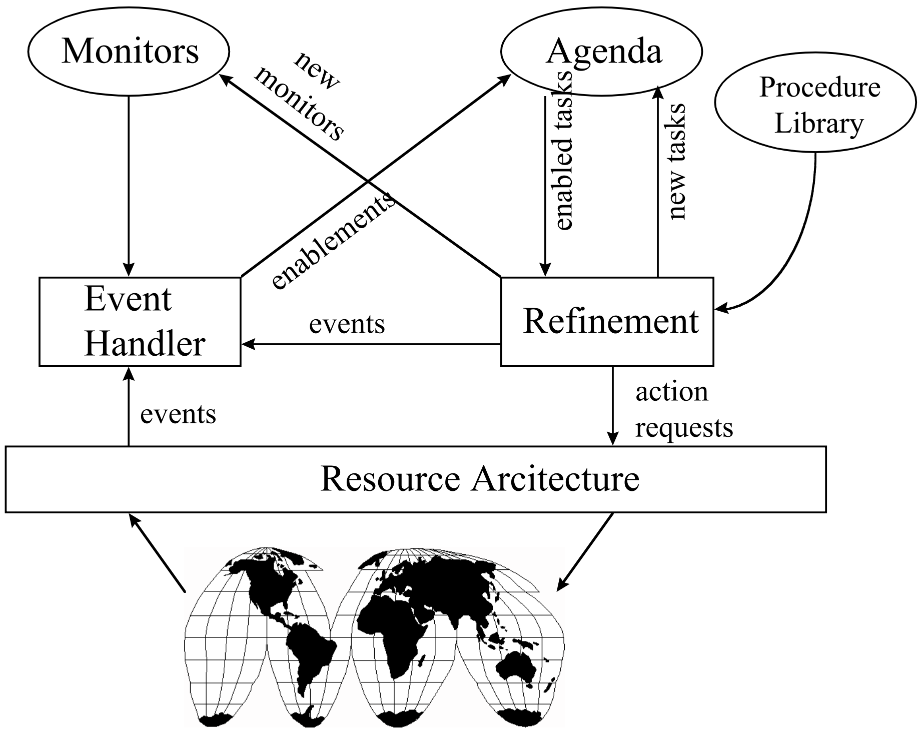


Figure 3.1 The APEX Sketchy Planner

3.1.2 Task execution

Some tasks designate low-level **primitive** actions. Once enabled, the planner executes a primitive by sending an action request to the resource architecture. This will usually initiate the specified action, although completion will normally require some time. **Task refinement** refers to the process of initiating execution of an enabled non-primitive task. The first step in refinement is to retrieve a procedure matching the task's index clause from the procedure library. Step clauses in the selected procedure are then used as templates to create new tasks.

For instance, the step labeled s1 in the above procedure creates a task to determine the identifying callsign for the specified aircraft (*plane-17* in this example) and bind this value to the variable ?*callsign*. This makes the information available to other tasks created by the procedure. Since the step does not designate any primitive action, achieving the new task entails refinement – i.e. finding a procedure whose index clause matches the form *(determine-callsign-for-plane ?plane),* creating any tasks specified by the retrieved procedure, adding those tasks to the agenda, and so on (see figure 3.2).

1. Retrieve procedure for enabled task
2. Create new tasks, one for each step in procedure
3. Add variable bindings from index to new tasks' local context
4. Create monitor for each new task precondition
5. Add new tasks to agenda
6. Add new monitors to monitors store

Figure 3.2 Task refinement

A task may require control of one or more resources in the resource architecture as a precondition for its execution. For example, each of the *say* tasks resulting from steps s2, s3 and s4 require the VOCAL resource (whose characteristics will be described in chapter 5). Multiple tasks needing the same resource at the same time create a **task conflict**. Chapter 4 describes the general approach and PDL constructs employed to manage task conflicts.

3.1.3 Precondition handling

Tasks defined by steps of a PDL procedure are assumed to be concurrently executable. When a particular order is desired, this must be specified explicitly using the **waitfor clause**. A waitfor clause defines one or more (non-resource) preconditions for task execution; ordering is accomplished by making one task's completion a precondition for the other's enablement.

In the example procedure above, all steps except s1 specify preconditions. Consequently, the newly created task {s1} corresponding to step s1 begins in an enabled state and can thus be executed immediately, while all the other tasks begin in a pending state. For example, step s2 includes the clause *(waitfor ?s1)*. This is actually an abbreviation; the expanded form for the clause is *(waitfor (terminated ?s1 ?outcome))* where *?s1* refers to the task created by step s1 and ?*outcome* refers to the value resulting from executing the task. Thus, step s2's waitfor clause specifies that {s2} should remain pending until an event matching the form *(terminated {s1} ?outcome)* has occurred.

When the refinement process creates a task, it will also create and store a **monitor** structure for each precondition specified in the task-step's waitfor clause. This makes it possible for the planner to determine when the precondition has been satisfied. In particular, when a new event occurs and is detected by the planner, the event's description is matched against all stored monitors. If a match between the event and a particular monitor is found, the planner removes the monitor from **monitors** store and checks its associated task to see whether any preconditions remain unsatisfied. If all are satisfied, the task's state is set to enabled, allowing immediate execution.

For example, when task {s1} to determine ?*plane's* callsign has been successfully completed, the planner will terminate it and generate an event of the form *(terminated {s1} success)*. A monitor for events of type *(terminated {s1} ?outcome)* associated with task {s2} will be found to match the new event, triggering a check of the {s2}'s remaining preconditions.

Since {s2} has no other preconditions, the planner changes the task's state from pending to enabled.

3.1.4 Cognitive events and action requests

The APEX action selection architecture (sketchy planner) only interacts with the world model through perceptual and motor resources in the resource architecture. The interface between the two architectures is conceptually quite simple. Perceptual resources produce propositional descriptions, and then make the information available to action selection by generating a **cognitive event**. For example, after visually detecting a new airplane icon (an icon is a type of visual object, or "visob"), the VISION resource might generate an event proposition such as[4]:

```
(new (shape visob119 aircraft-icon))
```

Procedures may be defined to create tasks that are triggered by such events. For instance, a procedure that includes the following step

```
(step s5 (shift-gaze-to-visual-object ?visob)
         (waitfor (new (shape ?visob) aircraft-icon))
```

would create a task that reacts to a proposition of the above form by shifting gaze to the newly detected plane icon. Most of the time, the vision component processes multiple attributes of a perceived object at a time, generating an event for each. For example, when VISION produces an event signaling the shape of the newly appearing plane icon, other co-occurring events would make available information about color, orientation, and motion. Chapter 5 describes this process in detail.

[4] As discussed in chapter 5, the visual resource signals shape information somewhat differently than this simplified example implies. In particular, the value of a shape proposition is always a list of shapes in descending order of specificity. For instance, (shape visob119 (plane-icon icon visob)) denotes three shape values for visob119 with the least specific describing visob119 as simply a visual object.

Just as the action selection component receives input from resources in the form of events, it outputs **action requests** to resources in order to control their behavior. Action requests are generated by the primitive task SIGNAL-RESOURCE. For example, a procedure step creates a task to move the LEFT hand resource to a specified object would include the following step:

```
(step s6 (signal-resource left (move-to ?object)))
```

The proper syntax for an action request is different for each resource. How the resource responds to a request depends on its permanent and transient properties as defined by the resource model. The current APEX implementation specifies several resource models that may be the target of an action request. These include: LEFT, RIGHT, VOCAL, GAZE, and MEMORY, all discussed in chapter 5.

3.1.5 Local and global context

Several PDL constructs, including the index and waitfor clauses, may contain variables. Bindings (values) for these variables can be obtained from three sources:

- local context
- global context
- memory

Every task shares with its siblings[5] a set of bindings called a **local context**. Bindings are added to the local context in several circumstances. First, when the refinement process matches

[5] Tasks generated at the same time by a single procedure are *siblings* of one another, *children* to the (*parent*) task whose refinement created them, and *descendants* of their common parent and all of its *ancestors*. Genealogical nomenclature is standard for describing task hierarchies, and will be used periodically in this document.

a task to index clauses of procedures in the procedure library, variables in the index clause are bound to portions of the task description. If the match is valid, these bindings are used to form the initial local context for tasks created by refinement. For instance, when the task {clear-to-descend plane-17 3500} is successfully matched against a procedure with index *(clear-to-descend ?plane ?altitude),* the tasks generated by this procedure start with a local context that binds the variable ?*plane* to the value *plane-17* and *?altitude* to *3500*.

Further additions to local context result from matches between events and monitors. For example, when an event of the form *(new (shape visob119 aircraft-icon))* is successfully matched against a monitor for events of type *(new (shape ?visob) aircraft-icon),* a variable binding linking *?visob* with *visob119* is created. The new binding is then added to the local context of the monitor's associated task. Finally, extensions to local context occur when tasks receive return values from their subtasks. For example, when the task generated by the step

```
(step s1 (determine-callsign-for-plane ?plane => ?callsign))
```

terminates, a return value will be generated and assigned to the variable *?callsign*.[6]

Global context refers to a small number of variable bindings defined by the resource architecture but maintained in the sketchy planner and accessible to all tasks without effort. For instance, the current resource architecture defines the variable ?*subjective-workload* which represents a heuristic estimate of the agent's current overall workload and is used by many procedures to decide between different courses of action.

Globally accessible information is also stored as propositions in a generic memory store. However, unlike the global context which may be accessed "for free," this store is part of the resource architecture – a resource called MEMORY representing semantic working memory [Baddeley, 1990] – and can only be interacted with via action requests (retrieval cues) and events (retrieval output). Separating memory from the sketchy planner in no way adds to the functionality of the system but makes it easier to model human characteristics of memory such as

[6] Return value binding is just a special case of monitor binding and exists primarily as a notational convenience. Its use also provides greater uniformity with the RAP system described in [Firby, 1989] which the APEX ASA is meant to adapt (for human modeling) and extend.

the amount of time required for retrieval, interference, and various forms of retrieval error. Chapter 5 discusses the MEMORY resource in detail.

3.1.6 Initialization

Procedures in the procedure library cannot be retrieved and executed except by some pre-existing task on the agenda. This restriction applies even to reactive behaviors that are supposed to lie idle until triggered by an appropriate event. For example, the ATC procedure for handling a newly appearing plane is retrieved when an event matching a monitor of the form *(new (shape ?visob plane-icon))* occurs, causing a pending task {handle-new-plane ?visob} to become enabled. Such reactive tasks (and initial, non-reactive tasks if any) are created at the beginning of a simulation run by a process called **root task initialization**.

Root tasks are created automatically by the sketchy planner during an initialization process at the beginning of a simulation. In particular, the APEX sketchy planner creates two root tasks: {built-in} and {do-domain}. The former task invokes a set of procedures corresponding to innate reactive behaviors. For example, the natural human response to a new object in one's visual field (an "abrupt onset" [Remington et al., 1992]) is to shift gaze to the location of that object. To model this tendency, the procedure indexed by {built-in} includes a step of the form *(step s3 (fixate-on-abrupt-onsets))* which is run without preconditions at initialization-time, thus invoking the following procedure (simplified):

```
(procedure
  (index (orient-on-abrupt-onsets))
  (step s1 (visually-examine ?visob)
    (waitfor (new (shape ?visob ?shape)))))
```

Procedures encoding "built-in" behaviors are a permanent part of the APEX procedure library. Procedures invoked by {do-domain}, in contrast, are created by an APEX user to model initial and reactive behaviors in a particular domain. In ATC, these include such tasks as handling newly arrived planes, responding to new emergency conditions, and maintaining situation awareness by repeatedly scanning the radar display. To specify procedures that should be executed at initialization-time, a user must specify a procedure with the index *(do-domain)* and steps corresponding to the selected procedures. For instance, an abbreviated version of the ATC do-domain procedure looks like:

```
(procedure
  (index (do-domain))
  (step s1 (handle-all-new-planes))
  (step s2 (handle-all-emergency-conditions))
  (step s3 (maintain-airspace-situation-awareness)))
```

The sketchy planner will automatically execute steps of the do-domain procedure before any events occur in the simulated world.

3.2 Procedure Definition Language

The central construct in the APEX procedure definition language (PDL), the **procedure**, and the basic PDL clauses needed to define a procedure have already been introduced. The next two sections describe the basic syntax in more depth and introduce a few additional language constructs. Section 3.4 will discuss how these language elements can be used to represent a variety of simple behaviors. Chapter 4 will discuss extensions to PDL needed to manage multiple tasks.

3.2.1 The INDEX clause

Each procedure must include a single index clause. The index uniquely identifies a procedure and specifies a pattern that, when successfully matched with the description of a task undergoing refinement, indicates that the procedure is appropriate for carrying out the task. The syntax for an index clause is simply

```
(INDEX pattern)
```

where the pattern parameter is a parenthesized expression that can include constants and variables in any combination. Thus, the following are all valid index clauses:

```
(index (handle-new-aircraft))
(index (handle-new-aircraft ?plane))
(index (handle ?new-or-old aircraft ?plane))
(index (?action new-aircraft ?plane))
```

The generic matching mechanism currently employed in APEX (from [Norvig, 1992]), includes a expressive notation for describing patterns. For example,

```
(index (handle-new-aircraft ?plane ?type (?if (not (equal ?type B-757)))))
```

specifies a match as long as the task description element corresponding to the variable *?type* does not equal *B-757*. Another useful matching constraint allows a variable to match 0 or more elements of the input pattern (the task description). For instance,

```
(index (say (?* ?message)))
```

means that the pattern matches any input list starting with the symbol "say." Subsequent elements of the list are bound as a list to the variable ?message. For a complete list of such pattern description elements, see [Norvig, 1992].

3.2.2 The STEP clause

Step clauses in a procedure specify the set of tasks which will be created when the procedure is invoked and may contain information on how to manage their execution. Each step must contain a step-tag and step-description; an optional output variable and/or any number of optional step clauses may also be added.

(STEP *step-tag step-description [=> {var|var-list}] [step-annotations]*)*

A *step-tag* can be any symbol, although no two steps in a procedure can use the same tag. These provide a way for steps in a procedure to refer to one another. When a procedure is invoked to refine a task, each resulting task is bound to a variable with the same name as the step that produced it. These are added to the local context along with the variable *?self* which is always bound to the task undergoing refinement.

The *step-description* is a parenthesized expression corresponding to the index of one or more procedures in the procedure library. When the step is used to create a task, the step-description is used as a template for the new task's task-description. For instance, the step *(step s5 (handle-new-plane))* will create a task of the form
{handle new-plane} which, when refined, will retrieve a procedure with index *(index (handle-new-plane)).*[7]

After the step-description, a step clause may include the special symbol => followed by a variable or variable list. These **output variables** become bound to the return value of a task upon its termination. For example, the following are valid step clauses:

```
(step a1 (determine-callsign-for-plane ?plane => ?callsign))
(step b1 (determine-callsign-and-weightclass => (?callsign ?weight))
```

Upon termination of a task generated by a1, the variable *?callsign* becomes bound to a value specified by the task's termination step (see section 3.2.5) and the binding added to the task's local context. A task generated by step b1 should return a two-item list, with the variables

[7] Task descriptions may contain variables. However, procedures must be constructed to insure that corresponding task-description variables are bound in the task's local context by the time the task becomes enabled. For example, (step s2 (say ?callsign)) would be valid if the variable ?callsign were included in a waitfor clause for s2, specified as a return value in a previous step, or named in the enclosing procedure's index clause.

?callsign and ?weight bound to successive list elements. This process allows earlier tasks to make needed information available for subsequent tasks.

Step annotations determine various aspects of task execution. They appear as clauses following the step description and output variables . The PDL step annotation clauses WAITFOR, PERIOD and FORALL are described in the remainder of this section; PRIORITY and INTERRUPT-COST are described in chapter 4.

3.2.3 The WAITFOR clause

A waitfor clause defines a set of task preconditions and causes tasks generated by the step definition in which it appears to start in a pending state. The task becomes enabled when **events**[8] corresponding to all specified preconditions occur. For example, a task created by

```
(step s5 (shift-gaze-to-visual-object ?visob)
    (waitfor (terminated ?s2 success) (new (visual-object ?visob))))
```

becomes enabled when task {s2} has terminated successfully and a new visual object has been detected. Disjunctive preconditions can be expressed by multiple waitfor clauses. For example, tasks generated by

```
(step s5 (shift-gaze-to-visual-object ?visob)
    (waitfor (terminated ?s2 success) (new (visual-object ?visob)))
    (waitfor (terminated ?s3 ?outcome)))
```

becomes enabled if {t2} has terminated successfully and a new object has been detected *or* if task {s3} terminates with any outcome. The simple task ordering construct *(terminated*

[8] Events are generated by perceptual components of the resource architecture or, in certain cases, by action selection mechanisms. In particular, the ASA generates termination events when tasks complete.

?<step> ?outcome) occurs often and can be abbreviated *?<step>*. Thus, the above step could also be represented

```
(step s5 (shift-gaze-to-visual-object ?visob)
     (waitfor (terminated ?s2 success) (new (visual-object ?visob)))
     (waitfor ?s3))
```

The general form of a waitfor clause is

(WAITFOR *pattern** [:AND *pred-function**])

Like patterns appearing in an index clause, waitfor patterns are represented using the conventions described in [Norvig, 1992]. These conventions can be used to constrain matches, for example by specifying that certain values cannot match a given variable or that two variables must match to the same value. Additional flexibility is provided by the optional :AND keyword which signifies that subsequent arguments are LISP predicate functions. If all events specified by waitfor patterns have occurred, predicates are evaluated; the task becomes enabled only if all predicate values are non-NIL.

3.2.4 The FORALL clause

Refinement usually creates one task for each step in a procedure. The forall clause causes a step to create one task for each item in a list. The general form of a forall clause is

(FORALL *variable1* IN {*variable2*|*list*)

which specifies that one task will be created for each item in a list (possibly represented as a variable). This capability is especially useful for processing groups of visual objects. For instance, the following step

```
(step s4 (examine-plane-icon ?plane)
    (forall ?plane in ?plane-icon-list))
```

creates one task to examine a plane icon for each icon in the list bound to ?plane-icon-list.

3.2.5 The PERIOD Clause

The period clause is used to define and control periodic (repetitive) behavior. The most common usage has form *(period :recurrent)* which declares that a task created by the specified step should be restarted immediately after it terminates; any preconditions associated with the task are reset to their initial unsatisfied state. For example, the following two steps cause the controller to periodically monitor two adjacent regions of the radar display. Both {s5} and {s6} are restarted when they terminate, but {s6} restarts in a pending state and must wait for {s5} before it can be executed.

```
(step s5 (monitor-display-region 8)
         (period :recurrent))
(step s6 (monitor-display-region 7)
         (period :recurrent)
         (waitfor ?s5))
```

The *:recurrent* argument takes an optional expression parameter which is evaluated when the task terminates (or at **reftime** – see below). If the expression evaluates to NIL, the task is not restarted. The full form of the period clause contains several optional arguments for controlling aspects of task periodicity.

(PERIOD [:RECURRENT [*expression*]] [:RECOVERY *interval*] [:ENABLED [*expression*]]
[:REFTIME {enabled|terminated}] [:MERGE [*expression*]])

As will be described in chapter 4, priority determines how strongly a task competes for needed resources with other tasks. The *:recovery* argument temporarily reduces a repeating task's priority in proportion to the amount of time since the task was last executed. This reflects a reduction in the importance or urgency of reexecuting the task. For example, there is little value to visually monitoring a display region soon after it was previously examined since little is likely to have changed; thus, early remonitoring is unlikely to be a good use of the GAZE resource. Altering the steps to declare (e.g.) 30 second recovery intervals allows action selection mechanisms to allocate this resource more effectively.

```
(step s5 (monitor-display-region 8)
        (period :recurrent :recovery 30))
(step s6 (monitor-display-region 7)
        (period :recurrent :recovery 30)
        (waitfor ?s5))
```

If the :*enabled* argument is present, a task is restarted with all of its waitfor preconditions satisfied. However, if the (optional) expression following the :enabled argument evaluates to NIL, preconditions are reasserted as usual.

The :*reftime* argument is used to specify when a recurrent task is restarted. By default, it is restarted when the previous task instance has *terminated*. Alternately, reftime can be set to *enabled*, creating a new instance of the task as soon as the previous instance becomes enabled. This option is especially useful for maintaining reactive responses. For example, the task generated from the following step causes the agent to shift gaze to newly appearing visual objects. When such an object appears, the task is enabled and a new instance of the task is created in pending state. This allows the behavior to handle objects that appear in quick succession. Thus, if an additional new object appears while shifting gaze to the previous object, the pending version of {s5} will become enabled.

```
(step s5 (shift-gaze-to-visual-object ?visob)
    (waitfor (new (visual-object ?visob)))
    (period :recurrent :reftime enabled))
```

The :*merge* argument makes it possible to suppress undesirable (redundant) periodicity arising from separate procedure calls. When the :merge argument is present, a newly created task is checked against all tasks on the agenda to determine whether an indistinguishable pre-existing task already exists. If so, the two tasks are combined so that the resulting agenda item serves both procedures. This capability is especially useful for eliminating redundant information acquisition tasks. For example, two tasks with the form {determine-altitude UA219}, each meant to determine the altitude of specified aircraft, should not both be carried out. If the procedure step from which the second instance of the task is derived specifies that such tasks should be merged, the second instance is not actually created. Instead, references to it from sibling tasks are made to point to the first instance and waitfor preconditions from the second are added to the first.

3.2.6 The ASSUME Clause

The rationale for the assume clause is intertwined with the idea of a routine procedure. Many procedures represent the result of an adaptive learning process in which an agent learns how to achieve goals effectively in a particular task environment. Procedures take advantage of regularities in the environment. To take a simple example, a procedure for obtaining a clean drinking glass in a household where dishes are regularly cleaned and stored away may involve searching in a particular cabinet. If dishes are cleaned infrequently, the routine procedure may be to find a dirty glass in the sink and clean it.

The ability to rely on assumptions (e.g. that dishes will be clean) rather than verify each precondition for a procedure's success, is a practical requirement for agents in realistic environments since verifying preconditions can be time-consuming, expensive, and sometime impossible. However, even very reliable assumptions may sometimes prove false. Experienced agents will learn ways to cope with occasional assumption failures in routine procedures. One type of coping behavior entails recognizing incidentally acquired information that implies an assumption failure and temporarily switching to a variation on a routine procedure to compensate.

The assume clause supports this kind of coping behavior. When embedded in a procedure, an assume clause declares that a specified proposition should be treated as an assumption. Whenever information contradicting the assumption becomes available to the sketchy planner, an indicator variable associated with the procedure is temporarily set to indicate that the assumption cannot be relied on. The general form of this clause is as follows:

(ASSUME *variable proposition duration*)

where the *variable* is always bound to either T or nil (true or false) and serves as an indicator of the validity of *proposition*. The clause creates a monitor (see section 3.1.3) for signals corresponding to the negation of *proposition*. When such a monitor is triggered, the specified *variable* is set to nil until a time interval equal to *duration* has passed.

For example, the routine procedure below is used when determining when the left runway is available for use by a plane on final landing approach. In particular, the procedure is invoked to decide whether to verify that the runway is available by retrieving information from working memory or rely on the high-frequency (default) assumption that the runway is available.

```
(special-procedure
  (index (select-specification-method determine-runway-availability left))
  (assume ?left-ok (available-runway left true) (minutes 20))
      (if ?left-ok ‘frequency-bias ‘retrieval))
```

Rather than verify each time a plane is to be guided onto a final approach, the procedure normally assumes that the left runway is available. However, if information to the contrary becomes available – e.g. if the controller is told of a closure by a supervisor – a monitor set up by the assumption clause in this procedure detects the condition and sets a variable *?left-ok* to *nil*. If the need to select a runway arises within the specified interval (20 minutes), reliance on the default will be suppressed, causing the agent to explicitly verify the assumption.

Since the normal way of carrying out a task is typically optimized for normal conditions, the agent should ideally resume reliance on the assumption as soon as possible. It is not always possible to know (or worth the effort to find out) when the normal situation has resumed, but most exceptions to assumptions underlying a routine procedure only tend to last for a characteristic interval. The duration parameter in the assumption clause allows a user to specify how long as assumption should be considered invalid after an assumption failure has been detected. Guidelines for setting this parameter will be discussed in chapter 6.

3.3 PDL Primitives and special procedures

The sketchy planner recursively decomposes tasks into subtasks or into any of the following non-decomposable **primitive** operations.

3.3.1 Terminate

A terminate step has the general form:

(TERMINATE *[task] [outcome] [>> return-value])*

The task created from such a step removes a specified task and all of its descendant subtasks from the agenda, removes monitors associated with these tasks from the monitors store, and generates an event of the form

(terminated *task outcome*)

The step's *task* parameter, often left unspecified, equals the terminated task's parent by default. Terminating this task means ending execution of all tasks created by the terminate step's enclosing procedure. The outcome parameter can be any expression and may include variables. Most frequently, it will equal **success** or **failure**, or else be left unspecified. In the latter case, the outcome success will be used by default.

Optionally, the terminate step may include an output value following the special symbol >>. This value is made available to the parent task if its step so specifies. Note that some parent tasks expect a list of specified length as a return value in order to bind each list element to a variable. If no output value is indicated, the default value NIL is returned.

3.3.2 Signal Resource

The signal resource primitive is used to request action by a specified cognitive or motor resource. Signal resource steps are specified with the following syntax:

(SIGNAL-RESOURCE *resource arg**)

where the number of arguments varies depending on the resource being signaled. For example, the VOCAL resource requires an *utterance* parameter, a list of words to be articulated, and optionally a *meaning* parameter representing how the should be interpreted by a hearer.

```
(procedure
  (index (say (?* ?message))
  (step s1 (parse-message ?message => (?utterance ?meaning)))
  (step s2 (signal-resource vocal ?utterance ?meaning) (waitfor ?s1))
  (step s3 (terminate)
    (waitfor (completed vocal ?utterance))))
```

For instance, in the above (simplified) version of the SAY procedure, task {s2} causes a signal to be sent to VOCAL specifying a list of words (the utterance) and their meaning. After VOCAL executes the utterance, it generates an event of the form

```
(completed vocal ?utterance)
```

which triggers termination of the say task. A description of the parameters required to signal other resources in the resource architecture is included in chapter 5.

3.3.3 Reset

The reset primitive is used to terminate and then immediately restart a specified task. It is especially useful for represent a “try again” strategy for coping with task failure. For example, the following step

```
(step s4 (reset ?s2)
     (waitfor (terminated ?s2 failure)))
```

causes task {s2} to be retried if it fails.

3.3.4 Generate Event

Events are generated automatically when new perceptual information becomes available, when a resource such as voice or gaze completes a “signalled” activity, and when a task terminates or changes state. Events can also be produced by the generate-event primitive which allows a procedure to specify events of any desired form. This capability has several applications for coordinating tasks generated by separate procedures. However, its primary purpose is to make observations inferred from perception available to active tasks, just as if they originated in perceptual resources.

For example, the following procedure causes a plane’s weight class – an attribute that partially determines minimum trailing distance behind another plane – from the color of its icon on a (hypothetical) radar display.

```
(procedure
  (index (determine ?plane weightclass from color))
  (step s1 (infer weightclass ?color)
    (waitfor (color ?plane ?color)))
  (step s2 (generate-event (weightclass ?plane ?color))
    (waitfor ?s1)))
```

When color information about the specified plane becomes available, its weightclass is inferred and the new information made available to all tasks (monitors) as an event.

3.3.5 Special Procedures

The SPECIAL-PROCEDURE construct allows users to define their own primitive actions, usually for the purpose of computing a return value. Like a normal procedure, the special procedure requires an index clause, and may also include assumption clauses if desired. Its general form is

(SPECIAL-PROCEDURE *index-clause [assume-clause*] body*)

where *body* can be any LISP expression and may include variables defined in the index or assume clauses. Like normal procedures, special procedures are invoked to carry out an enabled task whose description matches its index. However, executing a special procedure does not require time or control of any resources, and no preconditions may be placed on constituent activities. Instead, the invoking task is immediately terminated with the evaluation of the special procedure body as return value.

```
(special-procedure
  (index (infer weight-class ?color))
  (case ?color (white ‘small) (green ‘large) (blue ‘B757) (violet ‘heavy)))
```

The special-procedure construct is used mostly to compute output values, such as the outcome of running an inference rule. For example, the special-procedure above is used to infer a plane’s weight class from the color of its icon.

3.4 Basic issues in procedure representation

This section describes how the PDL syntax described in the previous section can be used to represent a variety of common behaviors.

3.4.1 Situation monitoring

In a changing and unpredictable environment, an operator must be prepared to detect and respond to new or unanticipated situations. These include not only new opportunities to exploit and new threats that must be averted, but routine events whose timing and exact nature cannot be determined in advance.

Passive monitoring refers to the process of maintaining a prepared response to some situation of interest. APEX carries out this kind of monitoring using monitor structures, generated on the basis of waitfor clauses (see section 3.1.3). For example, step s1 of the following procedure generates a monitor that will be triggered if perceptual processes detect a new aircraft on the radar display:

```
(procedure
  (index (handle-new-aircraft))
      (step s1 (decide-if-accept-handoff ?visob => ?decision)
            (waitfor (new-arrival ?plane)))
      (step s2 (act-on-handoff-decision ?decision ?visob) (waitfor ?s1))
      (step s3 (terminate) (waitfor ?s2))
```

The successful execution of such procedures depends on separately specified behaviors to insure that triggering events are detected in a timely fashion. In some cases, including that of handling new aircraft arrivals, timely and reliable detection results from routine "scanning" procedures used to maintain situation awareness. In other cases, reliable detection requires

active sensing – i.e. explicitly checking for a situation of interest. For example, an air traffic controller should verify that clearances issued to pilots have actually been carried out. The following procedure for verifying compliance with a recently issued descent clearance employs an active sensing strategy.

```
(procedure
  (index (verify-altitude-compliance ?plane ?initial-alt ?target-alt))
  (step s1 (determine-vertical-motion ?plane) => (?current-alt ?trend))
  (step s2 (compute-alt-status ?initial-alt ?current-alt ?trend ?target-alt => ?status)
     (waitfor ?s1))
  (step s3 (generate-event (compliance (alt ?plane ?target-alt) ?status))
     (waitfor ?s2)
  (step s4 (terminate) (waitfor ?s3))
```

The procedure uses basic PDL constructs to initiate sensing actions, interpret information thus acquired, and trigger a possible response. In particular, step s1 indexes a procedure to visually check *?plane*'s datablock for its current altitude and altitude trend (up, down or level). Step s2 invokes a procedure that interprets resulting information – either verifying compliance or returning a description of the compliance failure. For example, if the plane has not moved from its original altitude, task {s2} will set *?status* to *no-action*; if the plane has descended below the prescribed altitude, *?status* will be set to *(altitude-bust low).*

Finally, {s3} generates an event describing the result of verification, thereby triggering a response if some passive monitor matches the event. If *?status* = *no-action*, for instance, a monitor matching the form below will be triggered, thus enabling that monitor's associated task.

```
(compliance (alt ?plane ?target-alt) no-action)
```

Some procedures employ an **active monitoring** strategy which couples active sensing with steps for responding to detected situations. For instance, a controller may wish to redirect (vector) all planes approaching the LAHAB navigational fix towards the DOWNE fix. This behavior can be represented as follows:

```
(procedure
  (index (vector-lahab-to-downe))
      (step s1 (check-for-plane-at lahab => ?plane))
      (step s2 (issue-clearance (vector ?plane downe))
            (waitfor ?s1 :and (not (null ?plane))))
      (step s3 (terminate)
            (waitfor ?s1 :and (null ?plane))
            (waitfor ?s2)))
```

It is often useful to execute such tasks repeatedly in order to keep track of changing and developing situations. The period clause discussed in section 3.2.5 provides means for producing and managing repetitious (periodic) behavior.

3.4.2 Procedure selection

People often know more than one procedure for accomplishing a task. In PDL, the task of selecting between routine procedures is itself routine, and is thus handled by steps of a procedure. For example, the following procedure is used to give a clearance to an aircraft using either a two-way radio or a keyboard. The latter (keyboard) method is appropriate for planes with **datalink** equipment which, though still under development, should eventually be onboard all commercial aircraft.

```
(procedure
  (index (give-clearance ?clearance ?plane))
  (step s1 (select-clearance-output-device ?plane => ?device))
  (step s2 (give-clearance ?clearance to ?plane using ?device) (waitfor ?s1))
  (step s3 (terminate) (waitfor ?s2)))
```

In this case, task {s1} determines which output device will be appropriate for delivering a clearance to the target aircraft. The outcome of {s1} determines which of two procedures will be retrieved to refine {s2} for different {s1} outcomes. If {s1} sets *?device* equal to *radio*, a procedure with index matching (give-clearance ?clearance to ?plane using radio) will be retrieved. If *?device = keyboard*, a different procedure with index clause *(give-clearance ?clearance to ?plane using keyboard)* will be used instead.

Some formalisms such as GOMS [Card et al., 1983] and RAPs [Firby, 1987] use specialized syntax for deciding between alternative procedures. This can be convenient but can also be limiting. In PDL, procedure selection and other decision tasks can make use of the full PDL syntax and can employ cognitive, perceptual, and motor resources as needed to make the best choice.

3.4.3 Decision procedures

Routine decisions are handled by the APEX sketchy planner in a uniform manner with routine activities of other kinds -- i.e. as procedure-derived tasks which are begun, aborted, interrupted, resumed, retried, and terminated as needed to handle situational constraints and to coordinate resource use with other tasks. Routine **decision procedures** can be incorporated into the procedure for some more general task, as in the *give-clearance* example of the previous section, or represented in a separate procedure.

A decision procedure includes at least two different kinds of activities: (1) acquiring information and (2) computing a decision based on this information. For example, in the following procedure for deciding whether to use radio or keyboard to issue an air traffic control clearance, task {s1} acquires decision-relevant information about the aircraft targeted for the clearance, and {s2} computes the decision from this information.

```
(procedure
  (index (select-clearance-output-device ?plane))
  (step s1 (determine-if-datalink available ?plane ?subjective-workload => ?datalink))
  (step s2 (compute-best-clearance-delivery-device ?datalink => ?best-device))
  (step s3 (terminate >> ?best-device)))
```

The process of computing a decision often amounts to nothing more than employing a simple rule. For instance, step s2 may index a procedure such as the following

```
(special-procedure
  (index (compute-best-clearance-delivery-device ?datalink))
  (if ?datalink 'keyboard 'radio))
```

which outputs the value *keyboard* if datalink equipment is aboard the aircraft, and otherwise outputs *radio*. While computing a decision often requires only a single, very simple procedure, acquiring needed information may require numerous procedures since more than one kind of information may be needed and each there may be several ways to acquire each type. For instance, to determine whether datalink is available on a given aircraft, a controller could choose among several procedures:

- check equipment information on the plane's flightstrip[9]
- retrieve the information from memory (if previously determined)
- infer its presence or absence from knowledge of plane's weight class (small aircraft usually lack advanced equipment, larger aircraft usually have it)

Information acquisition procedures typically vary in how quickly they can be carried out and how reliable the resulting information is likely to be. In the above example, checking the flightstrip for equipment information is likely to be the slowest but most reliable method, while inference from weight information is fastest and least reliable. Typically, such speed accuracy tradeoffs are resolved partially on the basis of workload – in low or moderate workload, reliable methods are preferred, but in high workload, people will fall back on faster methods.

The model estimates its current workload[10] and assigns the value (1-10) to the globally accessible variable ?*subjective-workload.* In step s1 of the procedure for selecting a clearance output device, the step description

[9] Since Datalink is still under development, questions such as where to denote the presence or absence this equipment have not yet been decided. Placement on the flightstrip is unlikely, and included here only for illustration.

[10] It is unclear whether useful domain-independent methods for computing subjective workload are possible. Currently, the APEX user must design a procedure to periodically recompute this value. The ATC model presently uses a crude measure based on the number of planes being handled.

```
(determine-if-datalink available ?plane ?subjective-workload)
```

matches different procedure indexes for different values of *subjective-workload*. Thus,

```
(index (determine-if-datalink-available ?plane ?sw (?if (< ?sw 8))))
(index (determine-if-datalink-available ?plane ?sw (?if (>= ?sw 8))))
```

the first index above belongs to a procedure for reading the flightstrip and is used whenever workload is less than 8; the second belongs to a procedure for inferring the presence of Datalink from a plane's weightclass, and is used in relatively high workload situations (8 or above)[11].

Workload value range for a given procedure should approximate the likely consequences of adaptive learning – i.e. the ability of a skilled operator to choose among methods given a speed-accuracy tradeoff. How these ranges should be set will depend entirely on domain-dependent factors such as the (possibly varying) importance of accuracy and the amount of speed or accuracy advantage of one procedure over another. However, the following guidelines might be useful.

- procedures should be assigned non-overlapping[12] ranges ordered from slowest/most-accurate to fastest/least-accurate
- a procedure inferior to some alternative in both speed and accuracy should not be used
- the larger a procedure's advantage over adjacent competitors in either speed or accuracy, the larger its workload range will tend to be

[11] Normally, information needed to determine whether a procedure is appropriate should be represented only in the calling context (not in its index). Thus, a better way to differentiate these procedures would be to use descriptive symbols such as READ and INFER in place of the index variable *?sw*; the calling procedure would explicitly compute which of the procedures should be selected.

[12] These guidelines assume that only workload, accuracy, and speed are relevant. If other factors affect the usability or utility of a procedure, a somewhat more complex scheme is needed.

- the greater importance of speed (accuracy), the larger the range assigned faster (more accurate) procedures

3.4.4 Controlling termination

Most procedures specify that a task should terminate once all of its constituent subtasks have terminated. The flexibility to specify termination at other times, provided by the terminate primitive, makes it possible to model several important behaviors, including:

- early success testing
- race conditions
- task aborts
- decision control

Some procedures should be carried out only if the goal they are meant to accomplish has not already been achieved. For example, as described in the previous section, one way to determine whether a plane is outfitted with Datalink equipment is to infer its presence from the plane's weight class. However, there is no need to spend effort on this task if the information has already been acquired and stored in working memory (see chapter 5). The following procedure specifies **early success testing**. If equipment information is successfully retrieved, the task terminates; otherwise remaining procedure steps are executed.

```
(procedure
  (index (determine-if-datalink-available ?plane ?sw (?if (>= ?sw 8))))
  (step s1 (mem-retrieve-truthval  (equipped ?plane datalink ?truthval) => ?truthval))
  (step s2 (determine-color ?plane => ?color)
    (waitfor ?s1 :and (equal ?truthval ‘unknown)))
  (step s3 (infer weightclass ?color => ?weight) (waitfor ?s2))
  (step s4 (infer datalink ?weight => ?truthval) (waitfor ?s3))
  (step s5 (terminate >> ?truthval)
    (waitfor ?s4)
    (waitfor ?s1 :and (not (equal ?truthval ‘unknown)))))
```

Similarly, a procedure can define a **race condition** in which two subtasks try to achieve the same goal in different ways; the overall task terminates when either subtask succeeds. For example, another way to determine if an aircraft has Datalink equipment is to find and read the plane's paper flightstrip located on the flightstrip board. The following procedure specifies that such action proceed concurrently with an attempt to retrieve equipage from memory. When the needed information is acquired from either source, the determination task terminates.

```
(procedure
   (index (determine-if-datalink-available ?plane ?sw (?if (< ?sw 8))))
   (step s1 (mem-retrieve-truthval  (equipped ?plane datalink ?truthval) => ?truthval))
   (step s2 (determine-callsign ?plane => ?callsign) )
   (step s3 (visually-locate flightstrip ?callsign => ?strip) (watifor ?s2))
   (step s4 (read ?strip) (waitfor ?s3))
   (step s5 (terminate >> ?truthval)
        (waitfor ?s1 :and (not (equal ?truthval 'unknown)))
        (waitfor (equipped ?plane datalink ?truthval))))[13]
```

The same basic approach can be used to terminate a task for other – e.g. to **abort** a task when it becomes unachievable, unnecessary, or doable by some other, preferable means. A similar technique can be used to control information acquisition in decision-making. In particular, a decision process should terminate when either (1) additional information would not produce a better decision or (2) external demands require an immediate decision based on whatever information has already been acquired.

For example, the following procedure extends the previously discussed method for selecting a clearance output device to take account of one additional factor; in particular, it is only appropriate to use the keyboard for clearance delivery if the plane has Datalink equipment AND it is located in portions of airspace where Datalink clearances are allowed.[14] Thus the procedure incorporates two information acquisition steps: one to determine equipage and one to determine the aircraft's location.

[13] Information obtained by reading is made available via events from the visual resource – e.g. reading equipment data off a flightstrip would produce an event of the form (equipped ?plane datalink ?truthval).

[14] It takes pilots more time to respond to clearances delivered by Datalink than to those delivered by radio. In regions of airspace requiring rapid compliance, it is unlikely that Datalink clearances will be allowed.

```
(procedure
  (index (select-clearance-output-device ?plane))
  (step s1 (determine-if-datalink available ?plane ?subjective-workload => ?datalink))
  (step s2 (determine-airspace-region ?plane => ?region))
  (step s3 (compute-best-device ?datalink ?region => (?best-device ?done))
    (period :recurrent) (waitfor ?s1) (waitfor ?s2))
  (step s4 (terminate >> ?best-device) (waitfor ?s3 :and (equal ?done true))))
```

In this example, information acquisition tasks {s1} and {s2} proceed in parallel. When either of these completes, {s3} computes a best guess decision and a value for the variable ?*done* which indicates whether the decision task should wait for more information. For instance, if {s2} finishes first, returning a region value appropriate for Datalink clearances, {s3} returns *?best-device = keyboard* and *?done = false*. If instead, {s2} finishes first but returns a region inappropriate for Datalink, {s3} returns *?best-device = radio* and *?done = true*; in this case, the decision task terminates, returning a final decision.

3.4.5 Failure recovery

Human behavior is robust in the sense of being resilient against failure. People handle failure in variety of ways including trying again immediately, trying again later, aborting the task entirely, and explicitly diagnosing the cause of failure before selecting a new method to carry out the task. The best failure response will vary for different tasks and different failure indicators. For example, when a controller misspeaks while issuing a radio clearance, s/he will usually just repeat the utterance. In contrast, if the controller determines that a previously issued clearance has not produced the desired effect – i.e. the pilot has not complied with it – there will usually be an attempt to query the pilot for an explanation before deciding whether to amend, reissue, or forego the clearance.

Because appropriate failure responses can vary significantly, the APEX action selection architecture does not make any automatic or default response. Instead, appropriate responses must specified explicitly in procedures. For example, the following step specifies that when task {s4} fails, it should be repeated:

```
                    (step s5 (reset ?s4) (waitfor (terminate ?s4 failure)))
```

Some tasks should not be restarted repeatedly and, in any case, not until certain preparatory actions are taken. For example, the following variation of the step above retries task {s4} after first saying "oops!," and only if it has not already been retried more than once.

```
          (step s5 (say oops!)
                    (waitfor (terminate ?s4 failure) :and (< (number-of-resets ?s4) 2)))
          (step s6 (reset ?s4) (waitfor ?s5))
```

Sometimes failure indicates that a new way of carrying out the task should be tried. For instance the following procedure steps specify that a clearance should first be given using the keyboard; if that fails, the controller should next try the radio.

```
          (step s8 (give-clearance ?clearance using keyboard))
          (step s9 (give-clearance ?clearance using radio)
                    (waitfor (terminate ?s8 failure)))
```

Alternately, a procedure can specify that when a particular subtask fails, the whole task should be considered to have failed. This leaves the question of how to respond up to the parent task (the grandparent of the failing subtask).

```
          (step s8 (give-clearance ?clearance using keyboard))
          (step s9 (terminate failure) (waitfor (terminate ?s8 failure)))
```

As a final note, modelers should be alert to the consequences of not specifying failure handling behaviors in procedures. First, procedures that assume success when the simulated world allows failure can produce incoherent behavior, a poor model of human behavior in most circumstances. For instance, a person would not try to move to and click on a display icon after failing to grasp the mouse; instead, the failing grasp action would be detected and retried first. Second, failure can immobilize the agent while it waits for an indicator of success. For instance, a procedure that prescribes an attempt to grasp the mouse but makes no provision for failure might never terminate, instead waiting indefinitely for an event indicating that the mouse is in hand. The action selection architecture currently does not include any mechanisms that prevent an agent from "hanging" in this way. For agents in a realistically complex and dynamic task environment, the importance of making procedures robust against failure can hardly be overemphasized.

Chapter 4
Multitask Management

This chapter describes APEX capabilities and PDL notations related to the management of multiple tasks. The chapter has been designed to be readable on its own. In particular, some material covered previously is reviewed; also, illustrative examples are taken primarily from the everyday domain of driving an automobile rather than from the air traffic control domain emphasized in previous and subsequent chapters.

4.1 Resource Conflicts

A capable human operator model – i.e. one that can reliably achieve its goals in realistic environments – must be able to manage multiple tasks in a complex, time-pressured, and partially uncertain world. For example, the APEX human operator model has been developed in the domain of air traffic control. As with many of the domains in which human simulation could prove most valuable, air traffic control consists almost entirely of routine activity; complexity arises primarily from the need to manage multiple tasks. For example, the task of guiding a plane from an airspace arrival point to landing at a destination airport typically involves issuing a series of standard turn and descent authorizations to each plane. Since such routines must be carried out over minutes or tens of minutes, the task of handling any individual plane must be periodically interrupted to handle new arrivals or resume a previously interrupted plane-handling task.

Plan execution systems (e.g. [Pell, et al., 1997; Hayes-Roth, 1995; Simmons, 1994; Gat, 1992; Cohen, 1989; Firby, 1989; Georgoff and Lansky, 1988]), also called *sketchy planners* have been designed specifically to cope with the time-pressure and uncertainty inherent in these kinds of environments. The APEX sketchy planner incorporates and builds on multitask management capabilities found in previous systems.

The problem of coordinating the execution of multiple tasks differs from that of executing a single task because tasks can interact. For example, two tasks interact benignly when one reduces the execution time, likelihood of failure, or risk of some undesirable side effect from the other. Perhaps the most common interaction between routine tasks results from competition for resources.

Each of an agent's cognitive, perceptual, and motor resources are typically limited in the sense that they can normally be used for only one task at a time. [15] For example, a task that requires the *gaze* resource to examine a visual location cannot be carried out at the same time as a task that requires gaze to examine a different location. When separate tasks make incompatible demands for a resource, a *resource conflict* between them exists. To manage multiple tasks effectively, an agent must be able to detect and resolve such conflicts.

To resolve a resource conflict, an agent needs to determine the relative priority of competing tasks, assign control of the resource to the winner, and decide what to do with the loser. The latter issue differentiates strategies for resolving the conflict. There are at least three basic strategies (cf. [Schneider and Detweiler, 1988]).

Shedding: eliminate low importance tasks
Delaying/Interrupting: force temporal separation between conflicting tasks
Circumventing: select methods for carrying out tasks that use different resources

Shedding involves neglecting or explicitly foregoing a task. This strategy is appropriate when demand for a resource exceeds availability. For the class of resources we are presently concerned with, those which become blocked when assigned to a task but are not depleted by use, demand is a function of task duration and task temporal constraints. For example, a task can be characterized as requiring the gaze resource for 15 seconds and having a completion deadline 20 seconds hence. Excessive demand occurs when the combined demands of two or more tasks cannot be satisfied. For example, completion deadlines for two tasks with the above profile cannot both be satisfied. In such cases, it makes sense to abandon the less important task.

A second way to handle a resource conflict is to *delay or interrupt* one task in order to execute (or continue executing) another. Causing tasks to impose demands at different times avoids the need to shed a task, but introduces numerous complications. For example, deferring execution can increase the risk of task failure, increase the likelihood of some undesirable side-effect, and reduce the expected utility of a successful outcome. Mechanisms for resolving a resource conflict should take these effects into account in deciding whether to delay a task and which should be delayed.

Interrupting an ongoing task not only delays its completion, but may also require specialized activities to make the task robust against interruption. In particular, handling an interruption may involve carrying out actions to stabilize progress, safely wind down the interrupted activity, determine when the task should be resumed, and then restore task preconditions violated during the interruption interval. Mechanisms for deciding whether to interrupt a task should take the cost of these added activities into account.

The third basic strategy for resolving a conflict is to *circumvent* it by choosing non-conflicting (compatible) methods for carrying out tasks. For example, two tasks A and B might each require the gaze resource to acquire important and urgently needed information from spatially distant sources. Because both tasks are important, shedding one is very undesirable; and because both are urgent, delaying one is not an option. In this case, the best option is to find compatible methods for the tasks and thereby enable their concurrent execution. For instance, task A may also be achievable by retrieving the information from memory (perhaps with some risk that the information has become obsolete); switching to the memory-based method for A resolves the conflict. To resolve (or prevent) a task conflict by circumvention, mechanisms for

[15] The current approach applies to resources that block when allocated to a task, but not to those that may be concurrently shared (e.g. processing cycles) or those that are depleted by use (e.g. money).

selecting between alternative methods for achieving a task should be sensitive to actual or potential resource conflicts [Freed and Remington, 1997].

In addition to these basic strategies, conflicts can also be resolved by incorporating the tasks into an explicit, overarching procedure, effectively making them subtasks of a new, higher level task. For example, an agent can decide to timeshare, alternating control of a resource between tasks at specified intervals. Or instead, conflicting tasks may be treated as conjunctive goals to be planned for by classical planning or scheduling mechanisms. The process of determining an explicit coordinating procedure for conflicting tasks requires deliberative capabilities beyond those present in a sketchy planner. The present work focuses on simpler heuristic techniques needed to detect resource conflicts and carry out the basic resolution strategies described above.

4.2 Multitask management in APEX

Mechanisms for employing these multitask management strategies have been incorporated into the APEX architecture which consists primarily of two parts. The *action selection component*, a sketchy planner, interacts with the world through a set of cognitive, perceptual, and motor resources which together constitute *a resource architecture*. Resources represent agent limitations. In a human resource architecture, for example, the visual resource provides action selection with detailed information about visual objects in the direction of gaze but less detail with increasing angular distance. Cognitive and motor resources such as hands, voice, memory retrieval, and gaze are limited in that they can only be used to carry out one task a time.

To control resources and thereby generate behavior, action selection mechanisms apply procedural knowledge represented in a RAP-like [Firby, 1989] language called PDL (Procedure Definition Language) . The central construct in PDL is a **procedure** (see figure 4.1), each of which includes at least an **index clause** and one or more **step clauses**. The index identifies the procedure and describes the goal it serves. Each step clause describes a subgoal or auxiliary activity related to the main goal.

The planner's current goals are stored as **task** structures on the planner's **agenda**. When a task becomes enabled (eligible for immediate execution), two outcomes are possible. If the task corresponds to a primitive action, a description of the intended action is sent to a resource in the resource architecture which will try to carry it out. If instead, the task is a non-primitive, the planner retrieves a procedure from the **procedure library** whose index clause matches the task's description. Step clauses in the selected procedure are then used as templates to generate new tasks, which are themselves added to the agenda. For example, an enabled non-primitive task {turn-on-headlights}[16] would retrieve a procedure such as that represented in figure 4.1.

```
(procedure
  (index (turn-on-headlights)
  (step s1 (clear-hand left-hand))
  (step s2 (determine-pos headlight-ctl my-car => (?location ?orientation))
  (step s3 (grasp knob left-hand ?location ?orientation) (waitfor ?s1 ?s2))
  (step s4 (pull knob left-hand 0.4) (waitfor ?s3))
  (step s5 (ungrasp left-hand) (waitfor ?s4))
  (step s6 (terminate) (waitfor ?s5)))
```

Figure 4.1 Example PDL procedure

In APEX, steps are assumed to be concurrently executable unless otherwise specified. The **waitfor** clause is used to indicate ordering constraints. The general form of a waitfor clause is *(waitfor <eventform>*)* where eventforms can be either a procedure step-identifier or any parenthesized expression. Tasks created with waitfor conditions start in a **pending** state and become **enabled** only when all the **events** specified in the waitfor clause have occurred. Thus, tasks created by steps s1 and s2 begin enabled and may be carried out concurrently. Tasks arising form the remaining steps begin in a pending state.

Events arise primarily from two sources. First, perceptual resources (e.g. vision) produce events such as *(color visob-17 green)* to represent new or updated observations. Second, the sketchy planner produces events in certain cases, such as when a task is interrupted or

[16] APEX has only been tested in a simulated air traffic control environment. The everyday examples used in this chapter to describe its behavior are for illustration and have not actually been implemented.

following execution of an enabled **terminate** task (e.g. step s6 above). A terminate task ends execution of a specified task and generates an event of the form *(terminated <task> <outcome>)*; by default, <task> is the termination task's parent and <outcome> is 'success.' Since termination events are often used as the basis of task ordering, waitfor clauses can specify such events using the task's step identifier as an abbreviation – e.g. *(waitfor (terminated ?s4 success)) = (waitfor ?s4).*

4.3 Detecting Conflicts

The problem of detecting conflicts can be considered in two parts: (1) determining when particular tasks should be checked for conflict; and (2) determining whether a conflict exists between specified tasks. APEX handles the former question by checking for conflict between task pairs in two cases. First, when a task's non-resource preconditions (waitfor conditions) become satisfied, it is checked against ongoing tasks. If no conflict exists, its state is set to ongoing and the task is executed. Second, when a task has been delayed or interrupted to make resources available to a higher priority task, it is given a new opportunity to execute once the needed resource(s) become available – i.e. when the current controller terminates or becomes suspended; the delayed task is then checked for conflicts against all other pending tasks.

Determining whether two tasks conflict requires only knowing which resources each requires. However, it is important to distinguish between two senses in which a task can require a resource. A task requires *direct control* in order to elicit primitive actions from the resource. For example, checking the fuel gauge in an automobile requires direct control of gaze. Relatively long-lasting and abstract tasks require *indirect control*, meaning that they are likely to be decomposed into subtasks that need direct control. For example, the task of driving an automobile requires gaze in the sense that many of driving's constituent subtasks involve directing visual attention to one location or another.

Indirect control requirements are an important predictor of direct task conflicts. For example, driving and visually searching for a fallen object both require indirect control over the gaze resource, making it likely that their respective subtasks will conflict directly. Anticipated conflicts of this sort should be resolved just like direct conflicts – i.e. by shedding, delaying, or circumventing.

Resources requirements for a task are undetermined until a procedure is selected to carry it out. For instance, the task of searching for a fallen object will require gaze if performed visually, or a hand resource if carried out by grope-and-feel. PDL denotes resource requirements for a procedure using the PROFILE clause. For instance, the following clause should be added to the turn-on-headlights procedure described above:

```
(profile  (left-hand  8  10))
```

The general form of a profile clause is

(profile (<resource> <duration> <continuity>)).*

The <resource> parameter specifies a resource defined in the resource architecture – e.g. gaze, left-hand, memory-retrieval; <duration> denotes how long the task is likely to need the resource; and <continuity> specifies how long an interrupting task has to be before it constitutes a **significant interruption**. Tasks requiring the resource for an interval less than the specified continuity requirement are not considered significant in the sense that they do not create a resource conflict and do not invoke interruption-handling activities as described in section 4.5.

For example, the task of driving a car should not be interrupted in order to look for restaurant signs near the side of the road, even though both tasks need to control gaze. In contrast, the task of finding a good route on a road map typically requires the gaze resource for a much longer interval and thus conflicts with driving. Note that an interruption considered insignificant for a task may be significant for its subtasks. For instance, even though searching the roadside might not interrupt driving per se, subtasks such as tracking nearby traffic and maintaining a minimum distance from the car ahead may have to be interrupted to allow the search to proceed.

In the example profile clause, numerical terms used to denote time values (duration and continuity) follow a simple logarithmic scale. The value 9 corresponds to 1 second, 8 to 2 seconds, 7 to 4 seconds and so on. The continuity value 10 denotes two things. First, all interruptions to the task are considered significant. Second, a procedure containing such a profile clause may exercise direct control of a resource using the primitive step type SIGNAL-RESOURCE.

4.4 Prioritization

Prioritization determines the value of assigning control of resources to a specified task. The prioritization process is automatically invoked to resolve a newly detected resource conflict. It may also be invoked in response to evidence that a previous prioritization decision has become obsolete – i.e. when an event occurs that signifies a likely increase in the desirability of assigning resources to a deferred task, or a decrease in desirability of allowing an ongoing task to maintain resource control. Which particular events have such significance depends on the task domain.

In PDL, the prioritization process may be procedurally reinvoked for a specified task using a primitive REPRIORITIZE step; eventforms in the step's waitfor clause(s) specify conditions in which priority should be recomputed. For example, a procedure describing how to drive an automobile would include steps for periodically monitoring numerous visual locations such as dashboard instruments, other lanes of traffic, the road ahead, and the road behind. Task priorities vary dynamically, in this case to reflect differences in the frequency with which each should be carried out. The task of monitoring behind, in particular, is likely to have a fairly low priority at most times. However, if a driver detects a sufficiently loud car horn in that direction, the monitor-behind task becomes more important. The need to reassess it's priority can be represented as follows:

```
(procedure
  (index (drive-car))
     …
  (step s8 (monitor-behind))
  (step s9 (reprioritize ?s8)
     (waitfor (sound-type ?sound car-horn)
     (loudness ?sound ?db (?if (> ?db 30))))))
```

The relative priority of two tasks determines which gets control of a contested resource, and which gets shed, deferred, or changed to circumvent the conflict. In PDL, task priority is computed from a PRIORITY clause associated with the step from which the task was derived. Step priority may be specified as a constant or arithmetic expression as in the following examples:

```
(step s5 (monitor-fuel-gauge) (priority 3))
(step s6 (monitor-left-traffic) (priority ?x))
(step s7 (monitor-ahead) (priority (+ ?x ?y)))
```

In the present approach, priority derives from the possibility that specific, undesirable consequences will result if a task is deferred too long. For example, waiting too long to monitor the fuel gauge may result in running out of gas while driving. Such an event is a *basis* for setting priority. Each basis condition can be associated with an importance value and an urgency value. *Urgency* refers to the expected time available to complete the task before the basis event occurs. *Importance* quantifies the undesirability of the basis event. Running out of fuel, for example, will usually be associated with a relatively low urgency and fairly high importance. The general form used to denote priority is:

```
(priority <basis>  (importance <expression>) (urgency <expression>))
```

In many cases, a procedure step will be associated with multiple bases, reflecting a multiplicity of reasons to execute the task in a timely fashion. For instance, monitoring the fuel gauge is desirable not only as a means to avoid running out of fuel, but also to avoid having to refuel at inconvenient times (e.g. while driving to an appointment for which one is already late) or in inconvenient places (e.g. in rural areas where finding fuel may be difficult). Multiple bases are represented using multiple priority clauses.

```
(step s5 (monitor-fuel-gauge)
  (priority (run-empty) (importance 6) (urgency 2))
  (priority (delay-to-other-task) (importance ?x)  (urgency 3))
  (priority (excess-time-cost refuel) (importance ?x)  (urgency ?y)))
```

The priority value derived from a priority clause depends on how busy the agent is when the task needs the contested resource. If an agent has a lot to do (workload is high), tasks will have to be deferred, on average, for a relatively long interval. There may not be time to do all desired tasks – or more generally – to avoid all basis events. In such conditions, the importance associated with avoiding a basis event should be treated as more relevant than urgency in computing a priority, ensuring that those basis events which do occur will be the least damaging.

In low workload, the situation is reversed. With enough time to do all current tasks, importance may be irrelevant. The agent must only ensure that deadlines associated with each task are met. In these conditions, urgency should dominate the computation of task priority. The tradeoff between urgency and importance can be represented by the following equation:

$$\text{priority}_b = S*I_b + (S_{max}-S)U_b$$

S is subjective workload (a heuristic approximation of actual workload), S_{max} is the maximum workload value 9, and I_b and U_b represent importance and urgency for a specified basis (b). To determine a task's priority, APEX first computes a priority value for each basis, and then selects the maximum of these values.

For example, a monitor fuel gauge task derived from the procedure step s5 above includes three priority clauses. Assuming a subjective workload value of 5 (normal) the first priority clause

```
(priority (run-empty) (importance 6) (urgency 2))
```

yields a priority value of 38. Further assuming values of 4 and 5 for variables *?x* and *?y*, the clauses

```
(priority (delay-to-other-task) (importance ?x) (urgency 3))
(priority (excess-time-cost refuel) (importance ?x) (urgency ?y)))
```

yield values of 32 and 40 respectively. The prioritization algorithm assigns the maximum of these values, 40, as the overall priority for the monitor-fuel-gauge task.

4.5 Interruption Issues

A task acquires control of a resource in either of two ways. First, the resource becomes freely available when its current controller terminates. In this case, all tasks whose execution awaits control of the freed up resource are assigned current priority values; control is assigned to whichever task has the highest priority. Second, a higher priority task can seize a resource from its current controller, interrupting it and forcing it into a suspended state.

A suspended task recovers control of needed resources when it once again becomes the highest priority competitor for those resources. In this respect, such tasks are equivalent to pending tasks which have not yet begun. However, a suspended task may have ongoing subtasks which may be affected by the interruption. Two effects occur automatically: (1) descendant tasks no longer inherit priority from the suspended ancestor and (2) each descendant task is reprioritized, making it possible that a descendant will itself suffer interruption. Other effects are procedure-specific and must be specified explicitly. PDL includes several primitives steps useful for this purpose, including RESET and TERMINATE.

```
(step s4 (turn-on-headlights my-turn))
(step s5 (reset) (waitfor (suspended ?s4))
```

For example, step s5 above causes a turn-on-headlight task to terminate and restart if it ever becomes suspended. This behavior makes sense because interrupting the task, is likely to undo progress made towards successful completion. In particular, the driver may have gotten as far as moving the left hand towards the control knob at the time of suspension, after which the hand will likely be moved to some other location before the task is resumed.

4.5.1 Robustness against interruption

Discussions of planning and plan execution often consider the need to make tasks robust against failure. For example, the task of starting an automobile ignition might fail. A robust procedure for this task would incorporate knowledge that, in certain situations, repeating the turn-key step is a useful response following initial failure. The possibility that a task might be interrupted raises issues similar to those associated with task failure, and similarly requires specialized knowledge to make a task robust. The problem of coping with interruption can be divided into three parts: wind-down activities to be carried out as interruption occurs, suspension-time activities, and wind-up activities that take place when a task resumes.

It is not always safe or desirable to immediately transfer control of a resource from its current controller to the task that caused the interruption. For example, a task to read information off a map competes for resources with and may interrupt a driving task. To avoid a likely accident following abrupt interruption of the driving task, the agent should carry out a wind-down procedure [Gat, 1992] that includes steps to, e.g., pull over to the side of the road. The following step within the driving procedure achieves this behavior.

```
(step s15 (pull-over)
  (waitfor (suspended ?self))
  (priority (avoid-accident) (importance 10)  (urgency 10)))
```

Procedures may prescribe additional wind-down behaviors meant to (1) facilitate timely, cheap, and successful resumption, and (2) stabilize task preconditions and progress – i.e. make it more likely that portions of the task that have already been accomplished will remain in their current state until the task is resumed. All such actions can be made to trigger at suspension-time using the waitfor eventform *(suspended ?self).*

In some cases, suspending a task should enable or increase the priority of tasks to be carried out during the interruption interval. Typically, these will be either monitoring and maintenance tasks meant, like wind-down tasks, to insure timely resumption and maintain the stability of the suspended task preconditions and progress. Such suspension-time behavior will not inherit any priority from their suspended parent, and thus, like the pull-over procedure above, will normally require their own source of priority.

Wind-up activities are carried out before a task regains control of resources and are used primarily to facilitate resuming after interruption. Typically, wind-up procedures will include steps for assessing and "repairing" the situation at resume-time – especially including progress reversals and violated preconditions. For example, a windup activity following a driving interruption and subsequent pull-over behavior might involve moving safely moving back on to the road and merging with traffic.

4.5.2 Continuity and intermittency

Interruption raises issues relating to the continuity of task execution. Three issues seem especially important. The first, discussed in section 5.4, is that not all tasks requiring control of a given resource constitute significant interruptions of one another's continuity. The PROFILE clause allows one to specify how long a competing task must require the resource in order to be considered a source of conflict.

Second, to the extent that handling an interruption requires otherwise unnecessary effort to wind-down, manage suspension, and wind-up, interrupting an ongoing task imposes opportunity costs, separate from and in addition to the cost of deferring task completion. These costs should be taken account of in computing a task's priority. In particular, an ongoing task should have its priority increased in proportion to the costs imposed by interruption. In PDL, this value is specified using the INTERRUPT-COST clause. For example,

```
(interrupt-cost 5)
```

within the driving procedure indicates that a driving interruption should cause 5 to be added to a driving task's priority if it is currently ongoing.

The third major issue associated with continuity concerns *slack time* in a task's control of a given resource. For example, when stopped behind a red light, a driver's need for hands and gaze is temporarily reduced, making it possible to use those resources for other tasks. In driving, as in many other routine behaviors, such intermittent resource control requirements are normal; slack time arises at predictable times and with predictable frequency. A capable multitasking agent should be able to take advantage of these intervals to make full use of resources. In PDL, procedures denote the start and end of slack-time using the SUSPEND and REPRIORITIZE primitives.

```
(step s17 (suspend ?self)
  (waitfor (shape ?object traffic-signal)  (color ?object red)))
(step s18 (monitor-object ?object) (waitfor ?s17))
(step s19 (reprioritize ?self)
  (waitfor (color ?object green)))
```

Thus, in this example, the driving task will be suspended upon detection of a red light, making resources available for other tasks. It also enables a suspension-time task to monitor the traffic light, allowing timely reprioritization (and thus resumption) once the light turns green.

4.6 Computing Priority

To compute priority, APEX uses a version of the priority equation described in section 4.4 that takes into account four additional factors. First, an interrupt cost value IC, specified as noted, adds to a task's importance if the task is currently ongoing; when a task is not ongoing (pending or suspended), its IC value is 0. Second, urgency values increase over time, reflecting an increased likelihood of suffering some undesirable consequence of delayed execution. The priority computation uses the adjusted urgency value U_b'. Third, the priority of a repeating (periodic) task is reduced for an interval following its last execution if a refractory period is specified using a PERIOD clause (see section 3.2.5). The fraction of the refractory period that has passed is represented by the variable R. Finally, the priority equation recognizes limited interaction between the urgency and importance terms. For example, it is never worth wasting effort on a zero-importance task, even it has become highly urgent. Similarly, a highly important task with negligible urgency must be delayed to avoid the opportunity cost of execution. Such interactions are represented by the discount term 1/(1+x). Thus the final priority function:

$$\text{priority}_b = R\left[S\left(1-\frac{1}{U_b'+1}\right)(I_b + IC) + (S_{max} - S)\left(1-\frac{1}{I_b+1}\right)U_b'\right]$$

4.7 Multitasking improvements

APEX is part of an ongoing effort to build practical engineering models of human performance. It's development has been driven primarily by the need to perform capably in a simulated air traffic control world [Freed and Remington, 1997], a task environment that is especially demanding on an agent's ability to manage multiple tasks. Applying the model to ever more diverse air traffic control scenarios has helped to characterize numerous factors affecting how multiple tasks should be managed. Many of these factors have been accounted for in the current version of APEX; many others have yet to be handled.

For example, the current approach sets a task's priority equals the maximum of its basis priorities. This is appropriate when all bases refer to the same underlying factor (e.g. being late to a meeting vs. being very late). However, when bases represent distinct factors, overall priority should derive from their sum. Although APEX does not presently include mechanisms for determining basis distinctness, PDL anticipates this development by requiring a basis description in each priority clause.

Other prospective refinements to current mechanisms include allowing a basis to be suppressed if its associated factor is irrelevant in the current context, and allowing prioritization decisions to be made between compatible task groups rather than between pairs of tasks. The latter ability is important because the relative priority of two tasks is not always sufficient to determine which should be executed. For example: tasks A and B compete for resource X while A and C compete for Y. Since A blocks both B and C, their combined priority should be considered in deciding whether to give resources to A.

Perhaps the greatest challenge in extending the present approach will be to incorporate deliberative mechanisms needed to optimize multitasking performance and handle complex task interactions. The current approach manages multiple tasks using a heuristic method that, consistent with the sketchy planning framework in which it is embedded, assumes that little time will be available to reason carefully about task scheduling. Deliberative mechanisms would complement this approach by allowing an agent to manage tasks more effectively when more time is available.

Chapter 5
The Resource Architecture

As described in section 1.6, the APEX human operator model combines two components: an **action selection architecture** (ASA) that controls the simulated operator's behavior, and a **resource architecture** (RA) that constrains action selection mechanisms to operate within human perceptual, cognitive, and motor limitations. This chapter describes the basic elements of the RA.

The resource architecture contains two kinds of resources: **input resources** which convey information to the action selection architecture and **output resources** which produce agent behavior in accordance with ASA commands. Perceptual and memory functions are embedded in input resources – each with characteristic limiting properties that constrain the performance of the agent. As discussed in section 1.4.3, such properties include: temporal requirements for processing, precision limits, capacity limits, fatigue characteristics, and bias characteristics. Exemplifying each of these in turn, the vision resource take time to process new visual information, lacks the precision to make fine color discriminations, can only detect light in a certain range of the spectrum, becomes fatigued after lengthy exposure to bright light, and is subject to illusion.

Motor and cognitive functions are embedded in output resources, each with the limiting characteristics mentioned above. For example, the right hand resource takes time to carry out an action, has limited dexterity and strength, gets fatigued after continued action, and may be favored (over the left hand) for certain actions. Output resources have the additional limiting attribute "unique state" (see 1.4.3), a property that effectively restricts the use of an output resource to carrying out one task at a time. Conflict results when more than one task requires a resource at a given time; the action selection component incorporates capabilities for detecting and resolving these conflicts (see chapter 4).

Action selection controls an output resource by changing its state. For instance, the state of the right hand resource is partially characterized by the value of its **activity** attribute. The value of this attribute changes when the ASA can generate a resource signal (see section 3.1.4) – e.g., a signal may change the activity to
(grasp mouse), signifying that the hand is engaged in trying to grasp the mouse. Other aspects of state are set by the world or jointly by the ASA and the world. For instance, the **grasp** attribute specifies what object, if any, is currently being held. Its value is set partly by the ASA which must initiate a grasp activity, and partly by the world which must determine whether the grasp succeeds.

The currently implemented resource architecture consists of six resource components, including one input resource – VISION – and five output resources – GAZE, VOCAL, LEFT (hand), RIGHT, and MEMORY. The following sections describe each of these components and then discusses how additional resources are added to the APEX resource architecture.

5.1 Vision

The APEX vision model provides action selection with high-level descriptions of objects in the visual field. Human vision constructs such descriptions from "raw" retinal data. Though, in principal, one could use a computational model to simulate such a **constructive** process, this cannot be accomplished in practice since no existing computer vision model can approach human visual performance. Instead, APEX assumes that the simulated world represents the current **visual field** (all potentially available visual information) using high level terms that are meaningful to the ASA. The VISION component's function is to extract a subset of visual field representations and makes them available to action selection.

The purpose of this **extractive** model is to help predict human visual performance without having to emulate demanding and poorly understood computational processes underlying real vision. To take a simple example, one limitation of human visual performance arises because the eyes are located exclusively in the front (ventral) portion of the head; thus, a person cannot observe visual objects rear of the head's coronal plane. To model this limitation in an extractive framework, it is only necessary to know how many angular degrees an object lies from the forward visual axis. If an object's angular distance exceeds a threshold (about 90 degrees), its visual properties are not extracted and thus not made available to action selection.

5.1.1 The visual field

The APEX vision model assumes that the world simulator generates and maintains a visual field consisting of two kinds of structures: visual objects and regions. Each visual object is described by the following standard feature types: color, intensity, shape, orientation, size, position, motion-vector, and blink rate (see figure 5.1).

One interesting property of an object is the ease with which it can be discriminated from another object on the basis of one of its featural dimensions. Discriminability is usually measured in units of Just Noticeable Difference, or JNDs [McKee, 1981]. If the difference between two objects is below 1 JND, they cannot be distinguished by human perception along the given dimension. To assist in predicting object discriminability, values for all feature types except shape are represented quantitatively.

For example, VISION assumes a 2-value color representation (red-green and blue-yellow axes) [Stiles, 1946; Wyszecki and Stiles, 1967] in which a given distance between two points in color-space corresponds to uniform JND measure. Below a certain color-distance threshold corresponding to 1 JND, a human cannot detect a color difference. Intensity, orientation, size, position, motion, and blink rate are represented quantitatively using conventional measures.

These feature values can also be represented using a simplified, qualitative notation. For example, color can be denoted with color names such as *blue* and *green*. Similarly, orientation can be denoted as (e.g.) *horizontal* or *vertical* rather than in degrees from horizontal. It is much easier to specify simulated-world information displays with icon properties defined in everyday terms than in less well-known quantitative measures. Thus, using a qualitative notation trades the ability to predict discriminability for a certain amount of convenience. Eventually, it will be useful to augment APEX with software that simplifies the process of creating and specifying simulated world visual environments and makes it easier to employ the less convenient quantitative notations.

Shape is represented as a list of qualitative shape values in decreasing order of specificity. For example, on a hypothetical ATC display in which several plane-shaped icons are used to depict aircraft by weight class, a particular icon's shape might be represented by the list *(heavy-plane-icon plane-icon icon visob).* The visual system can provide action selection with more or less specific shape information in different circumstances. For example, when gazing directly at a visual object with the above shape characteristics, VISION would identify the object fully – i.e. as a *heavy-plane-icon*, a *plane-icon*, an *icon*, and a *visob*. However, if object is only observed in the visual periphery, VISION might describe its shape as simply a *visob*, undistinguished from any other visual item.

Feature Type	Feature Value Notation
Color	RG/BY color distance
Intensity	lumens
Shape	shape list
Orientation	degrees from horizontal
Size	cm - smallest distinct feature
Position	cm - 3D world-centered
Motion vector	cm/sec - 3D cartesian
Blink rate	hertz (default = 0)

Figure 5.1 Visual features and feature value measures

As mentioned in chapter 2, **regions** provide a usefully coarse way to represent location, allowing an agent, for example, to refer to the area lying "between the DOWNE and LAX fixes" on a radar display. Regions are essentially psychological constructs and therefore properly part of the agent model. However, for convenience, their boundaries are defined when specifying aspects of the agent's physical environment such as an information display (see section 2.3.2). In certain GAZE states, VISION can extract two kinds of region properties: the number of contained objects, and the set of objects with some specified attribute (e.g. color=green).

5.1.2 Perception-driven visual processing

The process of determining which aspects of the visual field should be made available to action selection takes place in two main phases, a perception-driven (bottom-up) phase and a knowledge-driven (top-down) phase. The perception-driven process determines a range of possible values for each feature of a visual object or region based on current perceptual information and the state of the gaze resource. For example, when directing gaze to a visual object, its color will be discriminated with relatively high precision (a narrower range of values) than if gaze were directed at some other object or to the entire region in which the object is contained.

VISION uses a separate precision-determination function[17] for each feature type. Each such function incorporates three limitations on visual performance. First, the human eye requires time to process a visual stimulus, in part because it takes time for the retina to gather enough light. To model this, determination functions take the amount of time GAZE has been fixed at the current location as a parameter. Second, visual acuity, the ability to discriminate fine-detail, decreases with visual angle for most features (although motion detection actually gets better in the periphery of vision). Thus, each function is provided the current GAZE direction (fixation value) as a parameter. Third, precision may vary depending on where attention, the internal component of gaze (see section 5.2), has been allocated; attention to an object or region is

[17] In principle, these functions could be used to model distortions (e.g. illusions) as well as variable-precision feature valuations.

sometimes a precondition for acquiring any value at all for some of its visual properties. Thus, attention locus is also provided as a parameter.

These three limitations on visual performance are considered in two steps. In the **pre-attentive** step, the model determines whether a given visual information item can be resolved without considering where the agent's attention is located. If so, the item is immediately passed to the top-down processing phase. Otherwise, the information item is processed by a second step representing the effects of **attentive** processing. If the item lies within the current attentional locus and this is sufficient to resolve the item (with greater than zero precision), it is passed on to top-down processing. Separating bottom-up processing into pre-attentive and attentive steps models the added time required to process attention-demanding information.

5.1.3 Knowledge-driven visual processing

After the perception-driven phase, imprecise property values may be narrowed on the basis of long-term knowledge or transient expectations. For example, an experienced user may know that only a few specific colors are used on a given display. When the first phase specifies that a particular object's color lies within a given region of color-space, the knowledge-driven phase narrows the range to include only those colors on the interface palette and in the specified color region. In many cases, especially when palette colors are well-chosen by the interface designer, only a single color-value will remain.[18]

5.1.4 Visual memory and output

The visual memory buffer stores the results of visual processing, allowing an agent to retain a representation of the visual field after gaze has shifted to a new location. New information items

[18] Procedures can be used in a similar way to exploit knowledge about feature value dependencies. For instance, if a blinking object is observed in the periphery of vision and it is known that only aircraft icons blink, a procedure could specifically infer and signal that the icon is an *aircraft*, even though VISION provides only the less precise value *visob* as its shape.

associated with an observed visual object are represented as a set of propositions – e.g. *(color visob-7 green)* – which together constitute a **visual object file** [Kahneman and Treisman, 1992]. Visual object files are automatically added to visual memory or updated at the end of each visual processing cycle.

The main purpose of an object file is to maintain a single representation of a visual object as it changes location and other properties, thus allowing an agent to focus on changes in the visual field. Accordingly, the model's visual component represents current visual information to the ASA in a way that depends on the contents of visual memory. For example, if VISION receives information of the form *(color visob-7 green)* while an identical proposition is available in memory, ASA will be informed *(refreshed (color visob-7 green))* meaning that a current observation has verified a known proposition. If instead VISION receives the same proposition when no information about the color of visob-7 is currently known, ASA will be informed of this with *(new (color visob-7 green))*.

VISION communicates with action selection mechanisms by generating events which may be detected by active monitors (see section 3.1.3). Such events have the form

(<memory-action> <proposition>)

where <proposition> describes a property of a visual object and <memory-action> describes the action taken to update visual memory as a result of the proposition. Memory action types include: new, refreshed, revalued, refined, and deleted. As indicated above, **new** indicates that the proposition adds information where none was available before; this will typically occur when an object is observed for the first time.

Refreshed indicates that previous memory information has been verified by current observation. **Revalued** means that the new information is incompatible with and supercedes the old information, e.g. when an object changes position or color. **Refined** means that the new information is more specific than but compatible with the old. For instance, an object's shape may previously have been identified as *(icon visob)*, but new observation may reveal it to be *(plane-icon icon visob)*. Finally, **deleted** means that information about the proposition is no longer available in the visual field – i.e. the object is no longer visible.

If the memory-action is any value except deleted, VISION generates an additional event of the form <proposition>. For example, if the current visual processing yields the proposition *(color visob-7 green)* when an identical proposition is already in visual memory, two events will be generated for the ASA: one of the form *(refreshed (color visob-7 green))* and one of the form *(color visob-7 green)*. This offers a simple way to specify waitfor conditions for visual events when the memory-effect is not important.

When all propositions resulting from a given cycle of visual processing are already stored in visual memory – i.e. all produce a refreshed event – the VISION component indicates that no new visual information has been produced by generating an additional event of the form *(nothing-new vision)*. This signals gaze control tasks in the ASA that a gaze shift may be warranted.

5.1.5 Example

This section illustrates the workings of the VISION resource using a simple example in which a new aircraft icon appears on the radar scope while gaze is oriented on a spatially distant location. The newly appearing visob is represented in the simworld's visual field by a set of propositions including:

(color visob-7 green)
(shape visob-7 (plane-icon icon visob))
(blink-rate visob-7 2)

VISION processes these propositions individually but concurrently. With gaze oriented far from the new object, perception-driven processing cannot determine the new object's color with any precision. Thus, the original proposition becomes *(color visob-7 ?any)* to reflect this lack of information. This structure is then passed to knowledge-driven processes. Since the radar display is known to employ multiple colors, this value cannot be further specified and is thus rejected – i.e. the proposition is not made available to the ASA. Similarly, the perception-driven processing phase cannot resolve the object's shape value to anything more specific than that it is a visual object. The proposition *(shape visob-7 (visob))* is then passed to the knowledge-driven phase which again cannot make it more specific. VISION then stores this limited shape information in visual memory and makes it available to the ASA by generating two events:

```
(new (shape visob-7 (visob)))
(shape visob-7 (visob))
```

The blink proposition is handled in the same way but with different results. Since motion information (including blink rate) can be resolved fairly well even for objects in the visual periphery, VISION delivers the proposition to the ASA intact. Moreover, since blink detection can be accomplished pre-attentively, the proposition circumvents the attention step of the perception-driven phase and is thus made available sooner than if attention had been required.

In response to the event *(new (shape visob-7 (visob)))*, the ASA enables a pending task that shifts gaze to any new, blinking object. With gaze now oriented on visob-7, VISION again processes the visual field propositions. This time, color information can be resolved, producing the following events

```
(new (color visob-7 green))
(color visob-7 green)
```

which are also encoded into visual memory. Similarly, VISION resolves shape information more specifically than was previously possible. However, since shape information for the object exists, the effects are somewhat different. In particular, the memory item *(shape visob-7 (visob))* is replaced with the more specific proposition (shape visob-7 *(plane-icon icon visob))*. VISION then informs the ASA with events of the form

```
(refined (shape visob-7 (plane-icon icon visob)))
(shape visob-7 (plane-icon icon visob))
```

thus informing the ASA that more specific shape information has become available.

5.2 Non-visual perception

Vision is the only perceptual function modeled explicitly in the current resource architecture. In place of explicit components for other perceptual faculties – including the auditory, kinesthetic, and proprioceptive senses – the current architecture provides a mechanism for passing information directly from the simulated world to the ASA. In particular, the LISP function *cogevent* causes a specified proposition to become available to the ASA at the beginning of its next activity cycle (see section 5.6).

Making perceptual information available without a mediating perceptual resource has at least two important consequences. First, an explicit perceptual resource allows perceptual processing to take time. In contrast, information conveyed by the cogevent function becomes available instantaneously. Second, a resource can be used to model limitations on perceptual capability. For instance, the vision resource models a decline in the ability to discriminate visual information located further from the center of fixation. The cogevent function makes all information available without limitation.

5.3 Gaze

In addition to the perceptual resources already described, the resource architecture includes several motor and cognitive resources. The GAZE resource represents two closely related human systems affecting visual processing. The **oculo-motor system** physically orients the eye, thereby directing the eye's fovea – a small region of densely packed retinal receptor cells capable of discriminating fine detail – towards a particular region of the visual field. The GAZE resource represents the selected direction, the **fixation** angle, as a pair of angular coordinates referred to as pan (rotation in the head's horizontal plane) and tilt (rotation in the median plane).

The **visual attention system** selects an object or region for enhanced perceptual processing, a prerequisite for determining certain visual properties. For example, the number of objects contained in a region can be approximated without counting them explicitly, but only if the region is the current locus of attention.[19] In the GAZE resource, the current attention state is represented by two values: an attentional **locus** which identifies an object or region of current interest, and an interest **attribute**.

The attribute parameter, set to NIL by default, is used primarily when searching a region for objects of interest. For example, to represent an interest in all green objects located in *region-3*, a task would set the locus parameter to *region-3* and the attribute parameter to *(color green)*. The attribute can be set to any single visual property including shape, even though shape is represented by a shape-list. VISION knows, for example, to interpret the attribute *(shape plane-icon)* as signifying an interest in visual objects with a shape property-value containing the symbol *plane-icon.* The interest parameter can also be set to a range using the keyword *:range*. For example, an attribute value of *(orientation (:range 0 45))* denotes an interest in visual objects whose orientation is known to fall (strictly) with the range 0 to 45 degrees.

[19] The oculo-motor system is usually treated as a slave of the attention system; when attention is shifted to a new object, the eye moves (saccades) to fixate on the object.

5.3.1 Controlling GAZE

As described in section 3.3.2, tasks control output resources using the primitive action *signal resource*. The form of an action specified in a signal resource step is

```
(<state-characteristic> <value>)
```

where a state characteristic is one of the components of the GAZE resource's state: *fixation, locus*, or *attribute*. The value parameter is a new value for the selected characteristic. For example, the following procedure step

```
(step s3 (signal-resource gaze (locus ?visob)))
```

specifies that the attention locus should be shifted to the visual object specified by the variable ?*visob*. Carrying out a signaled activity takes an amount of time specified by the resource. For instance, GAZE requires a constant[20] 50ms to complete an attention shift to a new locus.

GAZE automatically keeps track of how long the resource has been in its current state and signals this value at the end of each of its activity cycles with an event of the form *(gaze-held <time>)*. In conjunction with the event *(nothing-new vision)* which is generated by VISION to indicate that no new information was produced during the previous visual processing cycle, this time value can be used to decide when to shift attention and fixation to a new location. In particular, the innate procedure (see 3.1.6) *vis-examine* causes gaze to maintained on a given object or region until a specified minimum time has passed and VISION indicates no new information.

```
(procedure
  (index (vis-examine ?obj ?time))
  (profile (gaze 9 10))
  (step s1 (signal-resource gaze (locus ?obj)))
  (step s2 (terminate)
    (waitfor (nothing-new vision) (gaze-held ?time2 (?if (>= ?time2 ?time))))))
```

Using this procedure to examine an object allows the simulated agent to cope with the fact that information about a visual object tends to accumulate gradually while gaze is held on it. Another innate procedure causes gaze to shift when a new object appears in the periphery of vision.

```
(procedure
  (index (orient-on-abrupt-onset))
  (step s1 (vis-examine ?object)
  (waitfor (new (shape ?object ?shape)))
  (period :recurrent :reftime enabled)
  (priority 2)))
```

5.3.2 Searching

The psychological literature on searching for visual objects distinguishes two kinds of search: parallel and sequential. A parallel search involves concurrent visual processing of all objects in the visual field or in a region. The agent specifies some visual property that should uniquely identify the object being sought out. For example, a controller might know that a plane-icon of interest is the only red object currently on the radar display and search for it on that basis. Parallel searches may be carried out using the innate procedure below

[20] As discussed in section 2.6, the CSS simulation tool makes it possible to declare stochastically varying time requirements based on normal and gamma distributions.

```
(procedure
  (index (parallel-vis-search ?region for ?att))
  (profile (gaze 7 10))
  (step s1 (signal-resource gaze (locus ?region)))
  (step s2 (signal-resource gaze (attribute ?att)) (waitfor ?s1))
  (step s3 (terminate ?set)
    (waitfor (visob-set ?set ?att)))
```

which returns a possibly empty set (list) of visual objects located in the specified region and possessing the specified attribute. Note that there are limits on what kinds of visual properties can form the basis of a parallel search [Wolf, 1998; Wolf, 1994; Treisman and Sato, 1990]. The model approximates human limits by allowing only one search feature at a time, an approach based on findings by Treisman and Gelade [1980]. Thus one might search for green objects, or objects shaped like a plane-icon, but not green plane-icons.

Sequential search is used when parallel search based on a single feature cannot (or is unlikely to) discriminate the target of the search from distractor (non-target) objects. For example, to search for a green plane-icon, one might first identify the set of all plane-icons in the search region and then sequentially check whether each is colored green. The following procedure can be used to carry out such a search.

```
(procedure
  (index (sequential-vis-search ?region ?att1 ?att2))
  (step s1 (parallel-vis-search ?region ?att1 => ?searchset))
  (step s2 (verify-vis-attribute ?obj ?att2)
    (waitfor ?s1)
    (forall ?obj ?searchset))
  (step s3 (terminate >> ?target)
    (waitfor (has-attribute ?target ?att2)))
  (step s4 (terminate failure) (waitfor ?s2)))
```

The procedure verify-vis-attribute checks the designated visual object *?obj* for the specified attribute *?att2* and, if so, generates an event of the form:

```
(has-attribute ?obj ?att2)
```

If the target object is found, the search task terminates immediately, even if other objects in the search set have yet to be examined. If all objects are examined without finding the target, task {s2} will terminate causing the overall search task to terminate with failure.

5.3.3 Scanning

In many domains, maintaining situation awareness requires sequentially and repeatedly examining (scanning) visual locations of potential interest. For example, flying a plane requires continuously updating knowledge of situational information displayed on numerous instruments. Professional pilot's are taught a scanning procedure that prescribes an order in which instruments should be examined and insures that none will accidentally be omitted. Such a procedure can easily be represented in PDL

```
(procedure
  (index (scan-all-instruments))
  (step s1 (scan-instrument instrument-1))
  (step s2 (scan-instrument instrument-2) (waitfor ?s1))
  (step s3 (scan-instrument instrument-3) (waitfor ?s2))
    etc…)
```

and caused to execute repeatedly using a step of the following form:

```
(step <tag> (scan-all-instruments) (period :recurrent))
```

Air traffic controllers, like practitioners in most domains, develop their own strategies for sampling visually interesting objects and regions. Eye-tracking studies [Ellis and Stark, 1986] indicate that such strategies do not prescribe regular gaze shift sequences, making it impractical to predict individual transitions from one visual location to another. However, scanning behavior over longer periods is more predictable since visual locations tend to be examined with a regular frequency, roughly in proportion to the rate at which the location exhibits operationally significant changes.

Such scanning behavior can be modeled by procedures that specify unordered, recurrent visual-examination tasks for each region. Each such task is assigned a refractory period (time needed to recover full priority) proportional to the rate at which changes occur in that region. For example, the following procedure describes how a simulated controller might scan the northwest region(s) of LAX airspace:

```
(procedure
  (index (scan-northwest-arrival-sector))
  (step s1 (monitor-display-region hassa-arrivals)
        (period :recurrent :recovery 30)
        (priority timely-handoff-acceptance (urgency 4) (importance 3)))
  (step s2 (monitor-display-region hassa-to-geste)
        (period :recurrent :recovery 20)
        (priority avoid-region10-incursions (urgency 4) (importance 5)))
  (step s3 (monitor-display-region geste-to-downe)
        (period :recurrent :recovery 30)
        (priority avoid-region1-incursions (urgency 4) (importance 6))))
```

In accordance with prioritization computations discussed in chapter 4, such a procedure causes regions to be sampled at a rate proportional to their associated recovery times, adjusted for differences in base priority and to the overall number of regions to be examined.

5.3.4 Reading

Reading is currently modeled as a simple process of extracting individual words from a block of text. The simworld represents text using icons whose shape attribute contains the symbol textblock. For example, certain aircraft attributes including callsign, current altitude, and current airspeed are displayed as datablocks on the radar display (see 2.3.1). The ATC simworld represents the shape of a datablock with the value *(datablock textblock icon visob).* In addition, each textblock icon has a *text* attribute whose value is a list of words. For example, the text attribute-value of a datablock might be *(UA219 38 22)* signifying that the associated plane, United Airlines flight 219 is curently at 3800 feet elevation, traveling at 220 knots.

An APEX agent reads a textblock by shifting its attention locus to the textblock icon and setting the GAZE attribute parameter to the list position number of the word to be read. For instance, a datablock always contains the aircraft callsign as the first text item. Thus, the following procedure causes the agent to read the callsign from a specified datablock:

```
(procedure
  (index (read-datablock callsign ?datablock))
  (profile (gaze 9 10))
  (step s1 (signal-resource gaze (locus ?datablock)))
  (step s2 (signal-resource gaze (attribute 1)) (waitfor ?s1))
  (step s3 (terminate >> ?callsign)
    (waitfor (text ?datablock 1 ?callsign))))
```

Whenever the GAZE locus is set to a textblock and the attribute parameter to some index number, the VISION component generates an event of the form

```
(text <textblock> <index> <word>)
```

where <word> is the text item at position <index>. Procedure waitfor clauses can create monitors that detect such events and make the newly read word available to current tasks.

5.4 Voice

The VOCAL resource is employed by tasks to utter phrases (word sequences), currently at an unvarying rate of .25 seconds/word. To initiate an utterance requires a procedure step of the form:

```
(step <tag> (signal-resource vocal <phrase>))
```

Optionally, a step may use the form

```
(step <tag> (signal-resource vocal <phrase> <message>))
```

where the <message> parameter is used to inform the simworld of the phrase's meaning, thereby circumventing the need to have simworld processes employ natural language understanding to interpret uttered words.

The procedure *say*, included in the set of innate ASA procedures, simplifies the process of initiating an utterance. A procedure step simply specifies a list of words to be uttered, optionally followed by the keyword :*msg* and an expression defining the meaning of the phrase.

```
(procedure
    (index   (clear-to-descend ?plane ?altitude))
    (step s1 (determine-callsign-for-plane ?plane => ?callsign))
    (step s2 (say ?callsign) (waitfor ?s1)
    (step s3 (say descend and maintain flight level) (waitfor ?s2))
    (step s4 (say ?altitude :msg (clearance altitude ?callsign ?altitude))
            (waitfor ?s3))
    (step s5 (terminate) (waitfor ?s4)))
```

For example, the procedure above prescribes saying an aircraft's callsign, then the phrase "descend and maintain flight level" and then a specified altitude. The third of these three say actions also prescribes sending the simworld a message of the form

```
(clearance altitude ?callsign ?altitude)
```

indicating that the previously uttered words constitute a particular descent clearance to the plane identified by the variable ?*callsign*.

5.5 Hands

There are two hand resources, LEFT and RIGHT, each defined by three state attributes: location, grasp, and activity. A hand's location and grasp attributes – i.e. where it is and what, if anything, it is holding – determine what actions it can take. For example, the value of a hand's grasp attribute must be equal to *mouse* before it can use the mouse to move a display pointer. Similarly, it's location must equal *keyboard* in order to carry out a typing action. Action selection has direct control over a hand's activity attribute which describes what the hand is currently attempting. However, because hand motions can be obstructed and graspable objects can move, grasp and location attributes are determined jointly by action selection mechanisms and the simworld. For example, the following step sets the LEFT hand activity to *(grasp mouse).*

```
(step s4 (signal-resource left (grasp mouse)))
```

During each of a hand's activity cycles, arbitrarily set to 100ms[21], an event of the form

```
(activity <hand> <activity> <time>)
```

will be generated, indicating that the specified hand resource has been engaged in the named activity for an interval equal to <time>. This value can be used to terminate a hand activity that has gone on too long without success. For instance, the following step causes a task {s4} generated by the step above to terminate if it continues longer than 3 seconds.

```
(step s5 (terminate ?s4 failure)
        (waitfor (activity left (grasp mouse) ?time (?if (> ?time 3000)))))
```

[21] Every resource is associated with a cycle time that defines the shortest duration any modeled activity using that resource can take. Most activities will require multiple cycles. Cycle times are set to the largest feasible value in order to make efficient use of computing resources.

If the simworld determines that the task succeeds, it sets LEFT's grasp attribute to *mouse* and then indicates the new state by generating a perceptual event of the form:

```
(grasp mouse true)
```

Just as a grasp activity can change a hand's grasp attribute, a move activity can change its location attribute. Thus, a task derived from the following step

```
(step s4 (signal-resource left (move mouse)))
```

causes the LEFT hand to move to the mouse. Similarly,

```
(step s6 (signal-resource left (move mouse ?newpos)))
```

causes the left hand to move the mouse to the new position specified by *?newpos.* As with grasp actions, events are generated at each activity cycle to indicate how long a move activity has been going on. Success is determined by simulated world processes.

Note that the simulated world not only determines the result of a hand activity, but also determines when the result occurs. The APEX resource architecture includes a LISP function called *compute-Fitts-time* that can be used by world simulation mechanisms to approximate required time in accordance with Fitt's Law [Fitts and Peterson, 1964]. The Fitt's equation approximates the time required to move to a target location using only two parameters: the distance D from the current location to the target location, and the diameter S of the target. The resource architecture uses a form of the equation described in [Card, et.al., 1983]:

$$\text{Time (in milliseconds)} = 100(\log_2 (D/S + .5))$$

Procedures can specify two kinds of hand activities: single actions and action sequences. A single action activity is specified using the form (<action-type> <object>) such as *(grasp* *mouse)*. To specify an action sequence requires that the keyword *:sequence* follow the action-type specifier. For example, the following step requests that the RIGHT hand type a sequence of words:

```
(step s1 (signal-resource right (type :sequence (hi mom send money))))
```

A hand resource treats a sequential activity request as a succession of single actions. For instance, the step above is equivalent to the following steps:

```
(step s1a (signal-resource right (type hi)))
(step s1b (signal-resource right (type mom)) (waitfor ?s1a))
(step s1c (signal-resource right (type send)) (waitfor ?s1b))
(step s1d (signal-resource right (type money) (waitfor ?s1c))
```

Hand action sequences are similar to say actions carried out by the VOCAL resource. Like a say action, steps prescribing such a sequential hand action can specify a message to be sent to the simworld at the completion of the activity, signifying the overall meaning of the sequence. This prevents the simworld from having to parse the sequence to determine meaning. For example, the following step

```
(step s6 (signal-resource left
    (type :sequence (UA219 d 72) :msg (clearance altitude UA219 110)))
```

prescribes typing “UA219,” then “d,” and finally “110.” After completing these typing actions, LEFT informs the simworld that they should be interpreted as a clearance for an aircraft with callsign UA219 to change its altitude to 11000 feet.

5.6 Memory

5.6.1 Encoding

The MEMORY resource represents mechanisms for adding to and retrieving items from a simulated semantic working memory store [Baddeley, 1990]. The store is treated as an assertional database containing timestamped propositions. Steps of the following form are used to add (encode) new propositions.

```
(step <tag> (signal-resource memory (encode <proposition>)))
```

Alternately, a step can use the innate procedure *encode* which itself employs a step of the above form. For example, the following procedure step encodes a proposition indicating that the aircraft designated by *?callsign* has been issued a clearance to descend to *?altitude*.

```
(step s5 (encode (cleared altitude ?callsign ?altitude)))
```

Either method blocks the MEMORY resource for an interval currently set to a constant 100ms. The effect of an encode depends on whether the added proposition is of a sort that can become obsolete. For example, considering the following two pairs of propositions:

```
[1a]    (altitude UA219 120)
[1b]    (altitude UA219 110)
[2a]    (cleared altitude UA219 120)
[2b]    (cleared altitude UA219 110)
```

Propositions describing transient properties such as an aircraft's current altitude become obsolete when a new, incompatible proposition is encoded. For example, a plane cannot be both at 12000 feet (1a) and 11000 feet (1b) at the same time. Newly encoded proposition 1b thus supersedes stored proposition 1a. Propositions 2a and 2b represent altitude clearances issued to an aircraft at different times. A new clearance does not change fact that the previous clearance was issued; thus, 2b does not conflict with 2a.

A proposition that can become obsolete is a **fluent** [McCarthy and Hayes, 1969]. MEMORY provides the syntactic form below to declare that a certain class of propositions should be treated as fluents.

```
(declare-fluent <proposition-type> <var-list>)
```

The <proposition-type> is a list containing constants and variables; <var-list> identifies proposition-type variables used to determine whether two propositions match. For example, given the following fluent declaration,

```
(declare-fluent (altitude ?plane ?altitude) (?plane))

[a]    (altitude UA219 120)
[b]    (altitude UA219 110)
[c]    (altitude DL503 100)
```

propositions a and b match because values of the variable *?plane* listed in the <var-list> have the same value. Proposition c does not match either a or b because the value of *?plane* differs.[22]

Encoding a proposition causes MEMORY to generate cognitive events similar to those produced by VISION when items are encoded into visual memory. In particular, encoding produces events of the form

```
(<memory-event> <proposition>)
```

[22] Fluent declarations should be stored in the same file used to define procedures.

where <proposition> is the encoded item and <memory-event> can equal *refreshed, revalued,* or *new* depending on whether <proposition> exactly matches a current memory item, supercedes an existing (fluent) item[23], or has no match. Note that propositions are never retracted from memory, only superceded.

5.6.2 Retrieving

Retrieving information from the memory store requires blocking the MEMORY resource and then executing an action request derived from a step of the form:

```
(step <tag> (signal-resource memory (retrieve <cue>) [<maxage>]))
```

where the retrieval <cue> is a propositional form that may contain unbound variables, and optional argument <maxage> is a time value in milliseconds. If <maxage> is specified, only propositions encoded more recently than this value will be retrieved. Alternately (and equivalently) a step can employ the innate procedure *retrieve*. For example, a task arising from the following step

```
(step s2 (retrieve (cleared altitude ?callsign ?altitude)))
```

with *?callsign* bound to the value *UA219* causes MEMORY to retrieve any proposition matching *(cleared altitude UA219 ?altitude).* Similarly, the following step specifies a maximum age value 10000 and will thus only retrieve a proposition encoded during the preceding 10 seconds:

```
(step s5 (retrieve (altitude ?callsign ?altitude) :maxage 10000))
```

[23] Note that MEMORY only models one kind of interaction between new and old memory items – revaluing, when a new item supercedes a previous one. A complete model would represent other interactions such as proactive and retroactive memory interference (Baddley, 1990).

Executing a retrieve task causes the specified retrieval cue to be matched against all propositions in the memory store. Upon finding a match, MEMORY generates an event–describing the retrieved proposition. For example, the retrieval cue described in step *s2* above might match *(cleared altitude UA219 110)*, resulting in the following event:

```
(cleared altitude UA219 110)
```

In some cases, the retrieval cue will match multiple stored propositions. MEMORY retrieves the most recent proposition first. Unless the retrieval task is terminated or interrupted, MEMORY will then proceed to retrieve other matches in recency order, causing new events to be generated at intervals. When no more matches are found (or if none are found in the first place), MEMORY generates an event of the form:

```
(no-match <cue>)
```

Match (and match-failure) events may be detected by active monitors and used to determine when to terminate the retrieval attempt. Also, since the retrieval cue may contain unbound variables, the newly generated event may be used to form variable bindings. For example, the following procedure retrieves the most recently encoded value of a given plane's altitude and binds it to the variable ?*altitude*.

```
(procedure
    (index (determine-altitude ?callsign))
    (step s1 (retrieve (altitude ?callsign ?altitude)))
    (step s2 (terminate >> ?altitude)
      (waitfor (altitude ?callsign ?altitude))))
```

Retrieving an item from memory takes time. However, people can often determine that desired information is available in memory before it is successfully retrieved [cite feeling-of-knowing]. Conversely, people often recognize the futility of continuing a retrieval attempt after a brief effort. To represent this ability, MEMORY uses two different retrieval-time values – one for how long it takes to obtain a memory proposition and one for how long it takes to determine that none (or no more) will be forthcoming. MEMORY currently requires 1200ms to complete a retrieval (from [John and Newell, 1989]) and 600ms to determine retrieval failure.

5.7 Adding a new resource

The current set of resources model much of the perceptual and motor functionality humans require to carry out air traffic control tasks. However, applying APEX to new domains may require adding new resources which provide additional functionality. For instance, one may wish to add legs as a resource to allow the simulated agent to change location. Similarly, it may be desirable to replace current resource components with more scientifically accurate versions. For instance, the current VOCAL resource articulates words without error at a rate of 4 words per second. Existing scientific knowledge would support a more sophisticated model in which, for example, likelihood of speech error depends on similarity between closely spaced sounds, and the amount of time needed to articulate a word varies with such factors as word length and sentence structure.

5.7.1 New input resources

To simplify the addition or replacement of resource architecture components, APEX includes a standard software interface between the action selection architecture and the resource architecture. The standard interface for input resources consists simply of the LISP function

```
(cogevent <expression>)
```

which causes an event of the form <expression> to be made immediately available to the ASA. If there is no need to model the time required for perceptual mechanisms to process newly presented stimuli, the simworld can employ the *cogevent* function directly.

To model processing time requires two steps. First, the simworld must be made to maintain a **perceptual field** representing all detectable perceptual information[24] of the given type. Implementing a perceptual field entails simply creating a **store** component (see section 2.6) which takes input from the simworld **process**. Second, a process component representing perceptual processing mechanism and taking input from the perceptual field store must be added and assigned temporal properties. In particular, each such process has a characteristic **cycle time** denoting the interval required for a single unit of processing activity. Although the cycle time value has no theoretical status, the time course for the resource's processing actions can only be represented as multiples of this value. Cycle time must be set low enough to get whatever temporal resolution is desirable for the model but otherwise as high as possible to minimize simulation-slowing computational demands imposed at each activity cycle.

The process embeds a LISP function that makes perceptual field information available to the ASA. In the simplest case – when the model assumes that all information in the perceptual field should be made available after one cycle – this function will simply apply *cogevent* to all items in perceptual field. A more complex function is needed to model limits on perceptual detection capabilities and variability in time required for detection.

5.7.2 New output resources

An APEX user may also want to create new output resources which are controlled by the ASA. In the simplest case, the resource may only be needed to help model limits on human ability to carry out tasks concurrently. For example, the current model allows the LEFT and RIGHT hands to function with complete independence. In reality, using a hand for one task often

[24] Input resources include both perceptual functions such as vision and internal information gathering functions such as those used to retrieve information from a memory store. For clarity, this section only discusses how perceptual resources may be added; however, essentially the same approach works for all input resources.

precludes concurrent use of the other hand for a separate task, even when the activities are physically compatible. For example, most people would probably be unable to control two mouse pointing devices at the same time, or touch-type with one hand while using a mouse with the other.

Although the cause of such interactions is not completely understood, one way to model them is to use a new resource called FMC (fine-motor control) which must be owned by certain manual tasks (e.g. using a mouse, typing) as an execution prerequisite. This causes such tasks to conflict, thus requiring that they be carried out sequentially. Nothing special must be done to "create" such a resource. If it is named in the profile clause of a procedure, the ASA assumes a conflict with any other procedure that also names the resource (in accordance with conflict rules discussed in chapter 4).

Additional steps must be taken to add an output resource whose state can change as a result of a signal-resource task and which affects either the world (e.g. hands) or a cognitive process (e.g. GAZE affects VISION). First, one must define a store component to buffer action requests from the ASA. Second, a process component representing the new resource must be created and assigned a cycle time value; this component will also embed update function (see below) which carries out one cycle of resource activity. Third, a structure must be created that defines state parameters for the resource and stores their current values. For example, the structure representing GAZE has the following state parameters:

■ **fixation**	current orientation of the eye in angular coordinates
■ **locus**	identifies a currently attended visual object
■ **attribute**	identifies a visual attribute of interest (possibly NIL)
■ **t-held**	how long above state parameters have held their current value
■ **activity**	state change currently in progress (possibly NIL)
■ **t-start**	timestamp indicating when the current activity was initiated

Next, the LISP function *signal-resource* must be amended to allow signal-resource tasks to affect the new resource structure. For GAZE, this function was modified so that steps of the form

```
(step <tag> (signal-resource gaze (<parameter> <value>)))
```

would produce valid tasks as long as <parameter> equals either *fixation, locus*, or *attribute*, and <value> is a meaningful value for the respective parameter type. The effect of executing such a signal-resource task is to change the value of the GAZE activity parameter to equal *(<parameter> <value>)*.

A new output resource requires five additional functions to initialize its state, update its state as actions are carried out, and synchronize its activities with other cognitive processes and with the simworld. An **initialize-function** sets the resource's state values at the beginning of a simulation run. The **process-signal** function changes state values when a new signal is sent to the resource from the ASA. For instance, the gaze resource function *process-gaze-signal* takes two inputs, *<parameter>* and *<value>,* and sets gaze's activity parameter equal to *(<parameter> <value>)*. Thus, executing the task

```
{signal-resource gaze (locus plane-17)}
```

would result in the gaze activity parameter taking on the value *(locus plane-17).*

A **new-action** function (e.g. *new-gaze-action*) determines whether a new signal has just been sent to the resource and thus whether to initiate an activity cycle. A **pending-action** function determines whether the currently specified action is still ongoing. If not, the resource can stop cycling and thus save computational resources. GAZE is unusual in that it never stops cycling, even after a signaled activity has completed. Instead, each successive cycle produces a new event indicating how long the resource has been in its current state (the value of parameter t-held). Thus, the function *pending-gaze-action* always returns the value *T*.

An **update** function carries out one cycle worth of activity. The function update-gaze acts differently depending on which of three conditions holds. In the first case, the resource is still in the process of carrying out a specified action – i.e. simulating a saccade to a new fixation, shifting attention to a new locus, or setting an interest attribute. In this instance, the update function does nothing since it is waiting for the action to complete. Note that the update function must be able to determine how long an action will take and, using the t-start parameter, whether this interval has passed.

In the second case, the specified action has just completed. The gaze-update function sets the state parameters to the new value and generates an event to signal that the gaze action has completed. In the third case, when the currently specified activity completed on a previous cycle, *update-gaze* generates an event of the form *(held gaze <t-held>)* which tells the ASA how long fixation, locus, and attribute have been held their current values.

The **process-output** function is employed by "downline" mechanisms to access resource state values. For example, the VISION process uses the function process-gaze-output to retrieve current fixation, locus, and attribute values from GAZE. This function can also be used to inform the resource that the downline process has checked the information, a useful capability when synchronization between the two processes must be maintained or asynchronies handled.

Finally, some resources require a **set-state** function to allow a downline process to determine resource state values. This becomes necessary when the time requirement and outcome of a resource's actions must be determined jointly with the environment or mechanism its actions affect. For example, the LEFT and RIGHT hand resources can engage in grasp activities but their result is partially determined by the simworld which represents obstructions, object motion, slipperiness and other properties of the target of a grasp. The hand resources thus include a function *set-hand-state* which allows the grasp state (what the hand is currently grasping) to be set by the simulated world.

Chapter 6
Predicting Error

Human operator models have been used in diverse ways to guide and evaluate system design. For example, such models have been used to predict how quickly a trained operator will be able to carry out a routine procedure [Card et al., 1983; Gray et al., 1993], how quickly skilled performance will emerge after learning a task [Newell, 1990], how much workload a task will impose [Corker and Smith, 1993], whether the anthropometric properties of an interface (e.g. reachability of controls) are human-compatible [Corker and Smith, 1993], and whether information represented on a display will provide useful decision support [MacMillan, 1997].

The importance of predicting operator error at an early design stage has often been discussed in the human modeling literature [Olson and Olson, 1989; Reason, 1990; John and Kieras, 1994], but little progress has been made in constructing an appropriate operator model. This failure of progress stems primarily from a relatively weak scientific understanding of how and why errors occur.

> "..at this time, research on human errors is still far from providing more than the familiar rough guidelines concerning the prevention of user error. No prediction methodology, regardless of the theoretical approach, has yet been developed and recognized as satisfactory. [John and Kieras, 1994]"

Lacking adequate, scientifically tested theories or error (although see [Kitajima and Polson, 1995; Byrne and Bovair, 1997; Van Lehn, 1990]), it is worth considering whether relatively crude and largely untested theories might still be useful. This kind of theory could prove valuable by, for example, making it possible to predict error rates to within an order of magnitude, or by directing designers' attention to the kinds of circumstances in which errors are especially likely.

Everyday knowledge about human performance may offer a valuable starting point for such a theory. Though unable to support detailed or completely reliable predictions, a common-sense theory of human psychology provides a useful basis for certain predictions and is almost certainly better than nothing. In particular, common sense recognizes general human tendencies that often lead to error. By incorporating these tendencies into a model, we gain an ability to predict some of the most prevalent forms of human error and the circumstances in which these errors are especially likely.

For instance, it is commonly understood that people often omit "cleanup" activities that arise in service of a task but are not on the critical path to achieving the task's main goal. Examples include failing to replace a gas cap after refueling and forgetting to recover the original document after making a photocopy. Easily perceived reminders make these "postcompletion errors" [Byrne and Bovair, 1997] less likely, but they become more likely if other tasks are pressing. Risk increases further if subsequent tasks involve moving to a new physical environment lacking perceptual reminders of the unperformed activity. Knowing about qualitative influences on the likelihood of common error forms allows designers to assess the risk of certain usability problems and to evaluate possible interventions. For example, everyday knowledge about postcompletion errors allows a designer to predict that the operator of a newly designed device might plausibly forget to shut it off after use, and to consider interventions such as adding an ON/OFF light or making the device noisy while active.

As discussed in section 1.2 and 1.3, general knowledge about human performance is most easily applied in the context of specific and highly detailed scenarios. However, the amount of detail and the diversity of scenarios that need to be considered to evaluate a design increase dramatically as task environments become more complex, tasks become more prolonged, and the variety of possible operating conditions increases. Engineers can face great difficulty trying to predict usability problems with unaided common-sense, and may fail to predict problems that are easily understood from hindsight. An everyday understanding of human error incorporated into a human operator model could thus help direct designers' attention to usability problems that they might otherwise fail to consider.

6.1 Error Prediction Goals

A theory based primarily on everyday knowledge[25] of human psychology will almost certainly contain simplifications and omissions that limit its ability to predict operator performance. One way to set realistic predictive goals for a computer model that incorporates such a theory is to enumerate the kinds of predictions we would ideally like it to make, and then consider whether available knowledge is likely to prove adequate. Prospects for five error prediction goals are considered below.

1. Consequences of error
2. Likely forms of error
3. Circumstances in which a particular error type is especially likely
4. Likelihood of particular error type in given circumstance
5. Likelihood of error type in specified task environment

Figure 6.1 Types of error prediction in order of increasing difficulty

6.1.1 Predicting consequences of error

In addition to reducing error frequency, system designers need to prepare for operator error by facilitating their detection and minimizing their consequences. Predicting possible consequences of error does not require a detailed, accurate theory of how errors arise. Instead, the model can simply stipulate that certain operator actions will go awry a certain percentage of the time. Even if the assigned likelihood is wildly inaccurate, such a model allows the designer to consider the effect of error on system performance in different circumstances. For example, the model could be made to misspeak air traffic clearances 5% of the time, thus requiring the simulated controller to spend time restating the clearance. A designer might discover that in some cases, an inconveniently timed speech error forces the controller to either risk catastrophe by delaying repetition of the clearance, or else neglect some other important task. Such discoveries can help the designer determine whether appropriate safeguards are in place and whether high operator workload entails unacceptable risks.

6.1.2 Predicting likely forms of error

A somewhat more ambitious goal than that of predicting possible consequences of error is to anticipate what kinds of error are especially likely. For this purpose, everyday knowledge of human psychology is especially helpful. In particular, common sense recognizes a variety of distinct error tendencies that appear regularly in routine task environments. For instance, as illustrated by common errors such as forgetting to replace a gas cap after refueling and forgetting to retrieve one's bank card from an automatic teller machine, people have a tendency to neglect activities that must be carried out after the main task has been completed. Another well-known tendency is to carry out a task as one usually does it rather than as one intended to do it.

These tendencies can be characterized formally and incorporated into the model as decision-making **biases** [Reason, 1990]. Errors arising from an appropriately biased model would resemble human errors in two ways. First, the way in which the model's behavior deviates from correct behavior would be similar. For example, bias that sometimes favors routine over intended behavior would cause the model to occasionally make "capture errors" [Norman, 1981; Reason, 1990], a common form of human error that has been implicated in numerous accidents [Reason and Mycielska, 1982]. Second, the model's errors would occur in approximately the same circumstances in which human operators would make similar errors. For instance, errors would occur especially often when a non-routine action was intended.

[25] The emphasis on everyday knowledge is not meant to imply that scientific findings should be excluded from the model, only that in the absence of such findings, common sense theories can still be useful.

6.1.3 Predicting likely circumstances of error

Of course, people do not always err when trying to carry out a non-routine action; success is far more common. Error tendencies of all kinds tend to be balanced by cognitive and environmental factors that support correct behavior. The effect of environmental factors – e.g. the presence or absence of visual indicators that serve to remind an operator of correct behavior – are easily understood on the basis of everyday knowledge. Less insight is provided regarding the effect of mitigating cognitive factors. In particular, people seem to employ different mental strategies in different circumstances, with some strategies allowing greater possibility for error. For example, a person will sometimes mentally "rehearse" an intended action while waiting for an opportunity to carry it out, thus greatly increasing the chance of executing the correct action. At other times, the person might forego rehearsal, but explicitly consider whether the usual action is correct when deciding action. This also increases the chance that an unusual intention will be recalled. Alternately, people can act "automatically," allowing innate biases to guide behavior. At these times, decision biases play a strong role in determining action.

Common sense provides relatively little guidance for predicting what cognitive activities will be carried out, and thus whether innate biases will play an important role, in the process of making a decision. An alternative source of guidance comes from adopting a "rationality hypothesis" [Anderson, 1990] – i.e. an assumption that decision processes have adapted to become approximately optimal for the task environment. Cognitive activities such as mental rehearsal and recall from long-term memory are seen not only as useful for selecting correct actions, but also as demanding of limited cognitive resources. Biases are seen not only as a source of error, but also as useful heuristics that avoid the need to spend resources. To the extent that the rationality hypothesis holds, the question of whether bias or memory recall will determine the outcome of a decision task becomes a question of whether the expected marginal benefits of recall outweighs the expected costs. This approach will be considered in greater depth in section 7.2.

6.1.4 Predicting error likelihood

Reliable, quantitative estimates of error likelihood would be valuable to designers. For example, if a certain form of error would be troublesome but not especially dangerous in particular conditions, a designer may wish to know whether the likelihood of error in those conditions is great enough to justify expensive design interventions. Everyday knowledge and the rationality hypothesis provide a useful basis for predicting qualitative increases and decreases in error likelihood. But precise, quantitative knowledge needed to predict error likelihood requires precise performance measures that are not provided by everyday understanding and can be difficult to obtain scientifically.

For instance, consider the previously mentioned problem of forgetting to replace one's gas cap after refueling. Noticing the gas cap in an unattached position may serve to remind a driver of the unfinished task. So will explicitly reminding oneself to replace the cap while refueling, although this tactic becomes less effective as time since the last self-reminding act increases. Predicting the (quantitative) extent of these effects requires, e.g., a precise model of the processes that decrease self-reminding effectiveness over time and an ability to model agent motions in detail to determine whether the unattached gas cap comes into view. Neither everyday knowledge or current scientific understanding can provide a sufficiently complete and precise model.

Just as it would sometimes be desirable to know the likelihood of error in specified circumstances, it would also be desirable in some cases to know the overall likelihood (or frequency) of error in the task domain. Given a human operator model sufficient to satisfy the former goal, Monte Carlo simulation methods could be used to approximate the latter. For the time being, the goal of making quantitative predictions of specific or overall error likelihood should probably be considered out of reach.

6.2 A conceptual framework for error prediction

The approach to error prediction incorporated into APEX focuses on predicting the kinds of errors likely to arise in routine task environments, and on characterizing the circumstances in which such errors become especially likely. The conceptual framework behind this approach can be summarized as three main points:

1) Human cognition exhibits general biases that sometimes lead to error. These biases function as decision-making heuristics which make negligible demands on limited cognitive, perceptual, and motor resources. They can thus be seen as useful elements of resource-constrained cognitive processes.
2) Adaptive processes develop strategies which determine when resource-demanding activities will take place and when biases, which are less reliable but less demanding of resources, will be used instead. Over time, these strategies become approximately optimal for the task environment.
3) The same types of bias occur in different kinds of cognitive processing including perception, inference, memory retrieval, and action selection. Error in an early stage of cognitive processing (e.g. perception) can propagate, causing incorrect action in later stages and ultimately producing error.

6.2.1 Cognitive biases

Both scientific investigations and everyday psychological knowledge recognize a number of general tendencies, each of which can be characterized as a systematic bias on cognitive processes. Perhaps the most pervasive and most often investigated of these is sometimes called **frequency bias** [Reason, 1990] [26], the tendency to revert to high-frequency actions, beliefs, and interpretations. Such biases are general in two senses. First, they appear in a wide variety of human activities ranging from playing baseball to piloting an aircraft. Second, they affect many different cognitive processes including perception, memory retrieval, and action selection.

For example, frequency bias affects visual perception by influencing visual processes to see a routinely observed object as it normally appears, even when this differs from its actual current appearance. Similarly, frequency bias manifests in action-selection processes as a tendency to do the most frequently selected action, even at times when one had previously formulated an intention to do otherwise. In general, any cognitive process that can be characterized as selecting between alternatives – including alternative classifications of a stimulus, alternative interpretations of an observation, alternative responses, and so on – can be influenced by innate biases. In the remaining discussion, the term **decision-making** will be used to refer to cognitive activities that select between alternatives and may be subject to systematic bias.

APEX provides mechanisms for representing five different kinds of bias. As discussed frequency bias refers to the tendency for a decision process to choose the most frequently selected alternative. **Recency bias**, in contrast, favors the most recently selected alternative. **Confirmation bias** influences a decision-making process to select an expected value. An agent does not have to be consciously aware of an expectation for it to have an effect. For instance, auditory processes might expect a speaker to articulate vowels in a particular way based on observed vocal patterns ("accent"), even though most people could not even characterize these expectations.

Finally, decision-making may be biased by aspects of agent state such as hunger, fatigue, and workload (busyness). Of these, APEX currently models only **workload bias**, a factor that influences decisions with likely impact on workload towards alternatives that maintain a moderate level of busyness.

[26] Almost all of the ideas and perspectives presented in sections 6.21. and 6.2.2 appear in James Reason's book, *Human Error* [Reason, 1990]. These aspects of the APEX error prediction framework should be viewed largely as an adaptation and partial rearrangement of ideas expressed in that work.

6.2.2 Cognitive underspecification

To understand the role of bias in error, it is useful to consider how bias affects decision-making. Many decision processes can be characterized in terms of rules that select from alternative values based on a set of inputs. For instance, a typical action selection rule might be used to decide which of two routes to take when driving home from work. In the simple case in which route A is preferable during peak traffic and route B is preferable at all other times, the choice between A and B depends on three factors: the time of day, whether today is a weekend day, and whether today is a holiday. A rule expressing the desired decision-criterion might have the following form:

```
IF  it is currently “rush hour” AND it is a weekday AND it is not a holiday
THEN prefer route-A
ELSE prefer route-B
```

which can be represented by the APEX procedure below:

```
(special-procedure
  (index (select-route:work->home ?rush-hour ?weekday ?holiday))
  (if  (and ?rush-hour ?weekday ?holiday)
    route-A
        route-B))
```

To execute a rule, all of its input variables must be specified. Typically, the procedure that causes a rule to be carried out will also prescribe memory retrieval and perceptual scanning actions that add information to the current **decision-context**[27], thus making that information available to specify rule inputs.

Cognitive underspecification occurs when cognitive processes attempt to execute a rule while its inputs cannot be fully specified using information in the current context. In these cases, bias-derived values “fill in the blanks.” For example, the rule above for deciding a route home

[27] Since decisions in APEX result from the execution of explicit decision tasks, the decision context is simply the task context (see section 3.1.5).

from work might be invoked with the value for *?weekday* set to *true*, but leave values for the *?rush-hour* and *?holiday* variables unspecified in the current context. If a clock had recently been observed, recency bias might set *?rush-hour* to a previously inferred value; and since it is usually not a holiday when this rule is executed, frequency-bias might set *?holiday* to *false*.

Bias-derived values are often correct. For example, in predictable task environments where previously formulated expectations are often born out, confirmation bias will tend to provide reliable values. Similarly, frequency bias uses values which, by definition, have proven correct most often. Biases serve as useful heuristics when values from more reliable sources are absent. They also cause decision processes to err in predictable ways and in predictable circumstances. In the route selection task, for example, allowing frequency bias to determine whether it is currently a holiday rather than, e.g., recalling this information from memory, will sometimes lead to inappropriately choosing route A rather than route B.

6.2.3 The rationality principle

Viewing bias as a source of systematic error raises questions about how its effects should be modeled. In particular, constructing an appropriate model requires knowing:

- when a decision-rule will be underspecified, allowing bias to influence its output
- what kind(s) of bias and what bias value(s) will exist to specify a given rule input
- how biases will interact if multiple kinds apply

Answers in each case will depend partly on factors specific to the cognitive task and task environment. In the route selection task above, for example, the frequency bias value provided if *?weekday* is left unspecified will depend on whether it is usually a weekday when the agent drives home from work. The answers will also depend on general properties of human cognition such as how cognitive mechanisms determine which rule inputs to specify. Unfortunately, current scientific findings provide few constraints on how relevant aspects of cognition function.

In the absence of scientific guidance, an alternative way to address these issues is to apply a general principle articulated by Anderson [1990] that human cognitive processes adapt to the task environment to become approximately optimal.

> "*General Principle of Rationality*. The cognitive system operates at all times to optimize the adaptation of the behavior of the organism."

This approach[28] has been applied with some success to predicting performance in a number of cognitive systems including memory retrieval, causal inference, and categorization. Critiques of "rational analysis," not considered here, are discussed at length in [Anderson, 1990, ch. 1 and 6; Simon, 1991].

6.2.4 Rational decision making processes

Applied to the cognitive process of making decisions, the rationality principle implies that people will develop optimal decision-making strategies. This does not mean that the decisions themselves will necessarily be optimal from a normative standpoint, a possibility that has already been convincingly refuted [Tversky and Kahneman, 1974], only that the process of deciding will become optimal with respect to a set of agent goals, task environment regularities, and processing costs.

The first step in making this idea concrete [see Anderson, 1990 p.29] is to define decision-making and identify the goals of the decision-making process. In the current context, decision-making describes any cognitive action that selects between alternatives. This includes selecting from, e.g., alternative interpretations of a stimulus, alternative beliefs to infer from an observation, and alternative courses of action to follow in order to achieve a goal. Though

[28] Rational analysis is meant to guide the search for a scientific theory; hypotheses generated this way still depend on empirical testing to determine their validity. However, the main purpose of an engineering model of human behavior such as APEX is to get some idea of how human operators will perform before any empirical testing can be carried out. The model thus requires taking this method a step further – to generate assumptions about cognitive performance used to define the model's parameters.

decisions can be made using many different strategies (see [Payne,1993] for a particularly good review), most can be seen as involving two principal activities: (1) making decision-relevant information available to a decision-process by, for example, retrieving from memory, making inferences, or perceptually examining one's surroundings, and (2) executing a selection rule whose inputs are specified with information from step 1 when possible, or by bias when specifying information is unavailable.

Decision tasks in APEX are carried out on using procedures whose steps explicitly specify these two activities (see section 3.4.3). For instance, a procedure for deciding a route home from work might be represented as follows:

```
(procedure
  (index (decide-route home from work))
  (step s1 (determine-if-rush-hour => ?rush-hour))
  (step s2 (determine-if-weekday => ?weekday))
  (step s3 (determine-if-holiday => ?holiday))
  (step s4 (select-route:work->home ?rush-hour ?weekday ?holiday => ?route)
          (waitfor ?s1 ?s2 ?s3))
  (step s5 (terminate >> ?route) (waitfor ?s4)))
```

The main goal of a decision process is to select the alternative that best advances the agent's overall goals. However, several factors may entail using decision strategies that only yield satisfactory or likely-to-be-satisfactory decisions. One such factor is time; a satisfactory decision now is often more desirable than a better decision later. One way to control how long it takes to make a decision is to regulate time spent making decision-relevant information available. This may require trading accuracy for speed. For example, one might try to recall possibly obsolete information from memory rather than spend time visually searching for the information. Similarly, assuming that bias requires little or no time to specify a rule input, the best decision strategy may entail using bias rather than some more reliable but also more time-consuming method.

Another factor affecting how a decision process should be carried out is opportunity cost. In particular, employing **specification methods** such as memory retrieval and visual scanning to acquire decision-relevant information may require using limited cognitive or perceptual resources needed by other tasks. When resources needed to carry out one information specification method are in high demand, it may be desirable either to delay the decision task or to employ some other, possibly less accurate, method. APEX assumes that using bias to specify a decision-rule input requires no time or limited resources. Given this assumption, the best decision strategy will sometimes entail using bias rather than some more resource-demanding method, even though this risks occasional error.

Having specified the goals of decision-making – choose a good alternative, make a timely decision, and use limited resources effectively – it is now possible to use the principle of rationality to address the bias-related issues posed at the beginning of the previous section. The first issue is: when will a decision-rule be underspecified, allowing bias to influence its output? In light of the previous discussion, it should be clear that an optimal decision strategy must weigh the reliability advantage of resource-demanding methods against the time- and opportunity-cost advantages of letting bias specify an input. Thus, a given rule input should be left unspecified if, for all resource-demanding specification methods *m*,

$$\text{reliability-advantage}[m]*E(\text{cost-of-error}) < E(\text{time-cost}[m]) + E(\text{opportunity-cost}[m])$$

where *reliability–advantage* refers to the likelihood that *m* will yield the correct value minus the likelihood that bias will yield the correct value, and each expected cost term represents a cost measured against the agent's overall performance in the task domain.

To address the second and third issues – what types of bias will apply, and how will multiple biases interact? – it is useful to think of each form of bias as simply one possible method for specifying a decision-rule input. In some situations, the best method will require time and resources, while in others, some bias-based method will be best. An optimal decision-process chooses the best specification method in any circumstance; thus, any form of bias that provides the best specification method in some circumstances will be available. Biases will interact with one another in the same way they interact with resource-demanding methods – i.e. by competing as candidates for "best method."

The rationality hypothesis suggests that over time, decision-making processes will come to incorporate approximately optimal strategies for determining which method to use when specifying a decision-rule input. Optimal decision strategies will use bias in either of two ways depending largely on its reliability for specifying a given rule input. First, agents will learn to routinely use especially reliable bias, and to temporarily suppress its use in favor of more costly specification methods only in special cases. For instance, in selecting a route home, a driver may come to rely on frequency bias specifying that it is not a holiday; upon hearing that today actually is a holiday, reliance on the bias value will become temporarily suppressed. Second, when bias values are only moderately reliable, agents will generally learn to use reliable but expensive methods normally, but to fall back on bias values in conditions of high time-pressure and high workload.

6.2.5 Representing a decision-making strategy

In APEX, deciding how to specify a decision rule input is itself a decision task, carried out on the basis of a **meta-decision rule** incorporated into an explicit PDL procedure. Since allowing the execution of a **meta-decision procedure** to take time and limited resources would undermine its purpose (to help regulate time and resource-use in a decision-making task), such procedures should not prescribe the use of resource-demanding methods to specify meta-decision-rule inputs.[29]

For example, to determine whether it is currently a holiday when selecting a route home from work, decision processes can either retrieve this information from memory or rely on a high-frequency value specifying that it is not a holiday (*?holiday = false*). One plausible meta-rule for selecting between these specification methods is to use the high-frequency value (*false*) unless counterevidence for this value has recently been observed, in which case memory retrieval is used. Employing this strategy, one would tend to assume it is not a holiday. But after, e.g.,

[29] Meta-decision procedures are represented using the general procedure and special-procedure constructs defined in chapter 3. Thus, the proscription against resource-demanding specification methods is not enforced by APEX mechanisms, but is simply a guideline that should be observed by an APEX user.

being told otherwise, this tendency would become temporarily suppressed, causing one to verify the assumption by memory retrieval when deciding a route home rather than rely upon the assumption without further thought.

The following PDL procedures represent the above described strategy.

```
(procedure
  (index (determine-if-holiday))
  (step s1 (select-specification-method determine-holiday => ?method))
  (step s2 (determine-if-holiday by ?method => ?holiday) (waitfor ?s1))
  (step s3 (terminate >> ?holiday) (waitfor ?s2)))

(special-procedure
 (index (select-specification-method determine-holiday))
 (assume ?not-holiday (holiday today false) (12 hours))
 (if ?not-holiday
   'frequency-bias
   'memory-retrieval))
```

The construct *assume* (see section 3.2.6) can be used to represent events that should suppress reliance on the normal specification method. In particular, the clause specifies an assumption on which the correctness of the normal method depends. Detecting an event implying that this assumption has failed causes a variable specified in the *assume* clause to temporarily become set to the value *nil* (false). As an input to the meta-decision-rule, this variable serves to conditionalize suppression of the normal method.

The duration of suppression, declared as the third parameter in the *assume* clause, should be set to an approximately optimum value. Optimality in this case is a function of two main factors. First, as the likelihood that the normal situation has been restored increases, one should tend to revert back to the normal specification method which, by the rationality hypothesis, is presumably the best method in normal circumstances. The suppression duration should thus depend on either the typical duration of the unusual condition or on the rate at which one becomes reminded of the condition, whichever is shorter. Second, the more undesirable it would be to use the normal method incorrectly, the longer one should wait before reverting to it after observing evidence.

The meta-rule for deciding which method to use when specifying a decision-rule input must satisfy two conditions. First, its input values cannot come from time- and resource-demanding methods; only cost-free information such as bias, assume clause variables, and globals such as subjective-workload (see section 3.1.5) may be used. Second, it must closely approximate the theoretically optimum selection criterion which may be expressed as a minimization of the cost function

error-likelihood [m]*E(cost-of-error) + E(time-cost[m])+ E(opportunity-cost[m])

for all available methods *m*. Constructing a rule based on this function raises potentially difficult challenges. Because each term is an expected value, specifying that value requires knowing a great deal about the statistical structure of the environment. For example, to assign a time-cost to using memory retrieval requires knowing how much longer (on average) it will take for retrieval, what undesirable consequences can result from this delay, and how often each might be expected to occur. It seems reasonable to suppose that human adaptive processes develop estimates of such statistical regularities and use them to shape decision-making strategies. However, it is probably unrealistic to expect modelers to make detailed estimates of this sort, especially for complex task-environments that, containing devices and trained operators who do not yet exist, cannot be directly observed.

It is not yet clear how much this limits prospects for using models to predict bias-related error. One possibility is that human decision-making strategies are highly sensitive to the above parameters, making it unlikely that models can usefully predict when bias will influence decision-making. Another possibility is that rough, common-sense estimations will usually prove adequate. It should be clear that this will at least sometimes be the case. In specifying the variable ?holiday in the route selection task, the great reliability of the high-frequency value (?holiday = false) combined with the generally low cost of error (taking the less desirable route home), one could fairly assume that relying frequency-bias to specify this variable would, upon careful analysis, turn out to be the optimum decision strategy.

Chapter 7
Example Scenarios

The simple slips of action discussed in the previous chapter, usually harmless when they occur in everyday life, can have disastrous consequences when committed in domains such as air traffic control. While controller errors rarely result in an actual accident, their occurrence serves as a warning. The importance of **incidents** (as opposed to accidents) as indicators of latent flaws in system design is well-recognized in the aviation community. A database of incident reports called the Aviation Safety Reporting System [Chappell, 1994], currently containing over 100,000 incident entries, is often used to study the performance of the U.S. aviation system.

Such studies are sometimes used to justify changes to existing equipment and operating procedures. However, as discussed in chapter 2, making changes to already fielded systems can be very expensive. Significant modifications to aircraft equipment, for example, can involve retrofitting thousands of planes and retraining thousands of pilots on new procedures. Thus, the ability to predict error-facilitating design problems early in the design process can both prevent accidents and reduce the sometimes enormous cost of designing, fielding, and maintaining new systems.

APEX has been used to simulate controller performance in the current air traffic control regime and in possible future regimes involving equipment and procedures currently under development. Incidents that arise in these simulations can indicate significant design problems, particularly in components of the system's human-machine interface. The following sections describe three such incidents, all of which can occur in APEX simulations under certain conditions. These examples illustrate how human-system modeling can detect design problems that might otherwise be ignored.

7.1 Incident 1 – wrong runway

7.1.1 Scenario

At a TRACON air traffic control facility, one controller will often be assigned to the task of guiding planes through a region of airspace called an arrivals sector. This task involves taking planes from various sector entry points and getting them lined up at a safe distance from one another on landing approach to a particular airport. Some airports have two parallel runways. In such cases, the controller will form planes up into two lines.

Occasionally, a controller will be told that one of the two runways is closed and that all planes on approach to land must be directed to the remaining open runway. A controller's ability to direct planes exclusively to the open runway depends on remembering that the other runway is closed. How does the controller remember this important fact? Normally, the diversion of all inbound planes to the open runway produces an easily perceived reminder. In particular, the controller will detect only a single line of planes on approach to the airport, even though two lines (one to each runway) would normally be expected.

However, problems may arise in conditions of low workload. With few planes around, there is no visually distinct line of planes to either runway. Thus, the usual situation in which both runways are available is perceptually indistinguishable from the case of a single closed runway. The lack of perceptual support would then force the controller to rely on memory alone, thus increasing the chance that the controller will accidentally direct a plane to the closed runway.[30]

7.1.2 Simulation

The set of PDL procedures representing air traffic control "know-how" for the LAX airspace includes a procedure, described below, containing the clause:

[30] Examples of such incidents are documented in Aviation Safety Reporting System reports [Chappell, 1994] and in National Transportation Safety Board studies (e.g. [NTSB, 1986]).

```
(assume ?left-ok (available left-runway LAX true) (20 minutes))
```

Thus, whenever a cognitive event of the form *(available left-runway LAX false)* occurs, the model responds by setting the variable *?left-ok* to false for the specified interval of twenty minutes. Such a cognitive event occurs when the agent encodes the proposition into memory (see section 6.5) after being informed of the situation, and when the proposition is later retrieved from memory for any reason. In this example, the agent learns that the left runway is closed from an external source and encodes it, thus triggering an assumption violation and consequent binding of *nil* (false) to the variable *?left-ok*. Although this has no immediate effect on behavior, it can influence future decision-making.

Later, the simulated controller detects an aircraft approaching the DOWNE fix on its way to a landing at LAX. A task is initiated to select a runway for the plane and then issue clearances leading to a landing on that runway. The top-level procedure used to organize this response is represented as follows:

```
(procedure
  (index (handle-waypoint downe ?plane))
  (step s1 (retrieve (cleared ?plane ?callsign ?direct-to ?fix)
            (?if (member ?fix '(iaf-left iaf-right)))))
  (step s2 (select-lax-runway ?plane => ?runway)
            (waitfor (terminate ?s1 failure)))
  (step s3 (clear to initial approach-fix ?plane ?runway) (waitfor ?s2))
  (step s4 (terminate) (waitfor ?s3) (waitfor ?s1)))
```

In step *s1*, the agent determines whether it has already cleared the plane to one of the initial approach fixes, *iaf-left* or *iaf-right*. If so, the task terminates. Otherwise, the task proceeds by selecting a runway (*s2*) and then clearing the plane to the appropriate initial approach fix (*s3*). Executing the subtask corresponding to step *s2* requires retrieving and executing the runway selection procedure:

```
(procedure
  (index (select-lax-runway ?plane))
  (step s1 (determine-weightclass ?plane => ?weight))
  (step s2 (determine-runway-balance lax => ?balance))
  (step s3 (determine-runway-availability left => ?left))
  (step s4 (determine-runway-availability right => ?right))
  (step s5 (compute-best-runway ?weight ?balance ?left ?right => ?best)
           (waitfor ?s1 ?s2 ?s3 ?s4))
  (step s6 (terminate >> ?best) (waitfor ?s5)))
```

Using the terminology developed in the previous chapter, selecting a runway is a **decision task** carried out on the basis of the above **decision-procedure**. Steps s1 through s4 in the procedure acquire decision-relevant information and make it available in the current **decision context**. Step s5 executes a **decision rule** to produce a decision result.

The correctness of the resulting decision depends, in part, on whether correct information about the availability of the left runway is acquired upon executing {s3}. This information acquisition task proceeds in two steps based on the following procedure.

```
(procedure
  (index (detemine-runway-availability ?rwy))
  (step s1 (select-specification-method det-rwy-avail ?rwy => ?meth))
  (step s2 (execute-specification-method det-rwy-avail ?mth ?rwy => ?result)
     (waitfor ?s1))
  (step s3 (terminate >> ?result) (waitfor ?s2)))
```

The first step in determining runway availability is to run a **meta-decision-rule** to decide how runway availability will be assessed. Next the procedure executes the selected **specification method** and returns the resulting value (true if the runway is deemed available, false otherwise). The meta-decision-rule is represented using the procedure below. The rule normally prescribes reliance on **frequency bias**, but suppresses this reliance if a left runway closure has been detected or considered in the last 20 minutes.

```
(special-procedure
  (index (select-specification-method det-rwy-avail ?rwy))
  (assume ?left-ok (available-runway left true) (minutes 20))
  (if ?left-ok ‘frequency-bias ‘retrieval))
```

The selected specification method determines which of two procedures will be executed to compute a value for ?left (whether the left runway is usable) in the decision procedure above. If the frequency-bias method is selected, the following procedure will be carried out[31], returning the value *true*.

```
(special-procedure
  (index (execute-specification-method det-rwy-avail frequency-bias left))
  ‘true)
```

Otherwise, the procedure below will be used, causing a time- and resource-demanding memory retrieval action to be initiated.

```
(procedure
  (index (execute-specification-method det-rwy-avail retrieval ?rwy)
  (step s1 (retrieve (available-runway ?rwy false)))
  (step s2 (terminate >> false) (waitfor (terminate ?s1 success))
  (step s3 (terminate >> true) (waitfor (terminate ?s1 failure)))
```

In the described scenario, learning that the left runway has closed causes the agent to temporarily suppress reliance on frequency bias for assessing runway availability. Instead, for 20 minutes thereafter, the left runway’s availability is verified by memory retrieval whenever a runway selection task occurs. Eventually, this 20 minute suppression expires. When selecting a runway, the agent’s decisions will once again conform to the usual assumption. Other factors will then determine which runway is selected. For example, the controller may choose to direct a

[31] It is important to note that the use of explicit procedures for activities such as selecting and applying frequency bias does not imply that human cognitive mechanisms explicitly represent, or even explicitly carry out, such activities. As discussed in chapter 6, the APEX approach sharply distinguishes theoretically meaningful activities and constructs in the resource architecture from theoretically neutral activities and constructs in the action selection architecture. This separation has important practical advantages. For example, allowing the selection and application of frequency bias to be represented explicitly allows these events to appear in the simulation trace; mechanisms for analyzing the trace can thus refer to these events when explaining why an error occurred.

heavy plane to the longer left runway which, in normal circumstances, would allow the plane an easier and safer landing. With the left runway closed, actions following from this decision result in error.

Avoiding error requires maintaining suppression of inappropriate reliance on frequency bias for as long as the runway closure persists. In a variation of the described scenario in which no error occurs, visually perceived reminders of the runway closure cause suppression to be periodically renewed. In particular, whenever the agent attends to the landing approach region on the radar display (region 4), a pending procedure for detecting imbalances in load on the two runways is enabled and executed.

```
(procedure
 (index (determine-runway-balance lax))
 (step s1 (vis-examine 4left 500))
 (step s2 (vis-examine 4right 500)
      (waitfor (count 4left ?left-count)))
 (step s3 (compute-balance ?right-count ?left-count => ?balance)
      (waitfor (count 4right ?right-count)))
 (step s4 (generate-event (skewed-runway-usage lax ?balance))
      (waitfor ?s3 :and (> (abs ?balance) 2)))
 (step s5 (generate-event (imbalanced-runway-usage lax ?left-count ?right-count))
      (waitfor ?s3 :and (and (> (abs ?balance) 2)
(or (zerop left-count) (zerop right-count)))))
 (step s6 (terminate >> ?balance)
      (waitfor ?s5) (waitfor ?s4) (waitfor ?s3 :and (<= (abs ?balance) 2))))
```

Detecting an imbalance in the use of the two runways triggers an attempt to explain the situation in terms of known possible causes, or **error patterns** (see [Schank, 1986]). Two explanations are possible: careless assignment of runways to planes and runway closure. Since the rate at which planes take off and land is limited primarily by runway availability, explaining the imbalance as a result of carelessness helps direct attention to an important problem and should lead to more careful decision-making on subsequent runway decision tasks. Similarly, explaining the imbalance as a result of runway closure reminds the agent of an important constraint on runways selection and should make the agent more likely to consider that constraint on deciding on a runway. The following procedure performs the explanation task.

```
(procedure
  (index (explain runway-imbalance lax 0 ?number))
  (step s1 (retrieve (available-runway left false)))
  (step s2 (generate-event (careless det-rwy-balance true))
     (waitfor (terminate ?s1 failure)))
  (step s3 (terminate) (waitfor (terminate ?s1 success)) (waitfor ?s2)))
```

Executing this procedure will result in the generation of one of two signals depending on which explanation is selected. If a proposition of the form (available-runway left false) is retrieved in step s1, a cognitive event of that form will be produced. This refreshes the suppression of frequency-bias in determining runway availability, thus preventing error for at least the next 20 minutes.

7.1.3 Implications for design

Simulations of such incidents can draw attention to design problems that might otherwise have gone undetected, leading to useful changes in equipment, procedures, or work environment. Several methods may be employed to identify changes that might reduce the likelihood of operator error and thus prevent undesirable incidents from occurring. One method is to enforce the assumptions on which an experienced agent's normal decision-making strategies depend. In this example, that would mean insuring that runways are always available.

Alternately, and only somewhat more feasibly in this case, one might try to enforce the assumptions upon which an agents decision-making strategy depend. The error in this example arose because a scarcity of aircraft on the radar display deprived the agent of a useful perceptual indicator. Thus, insuring a steady flow of aircraft into the simulated controller's airspace would reduce the likelihood of error.

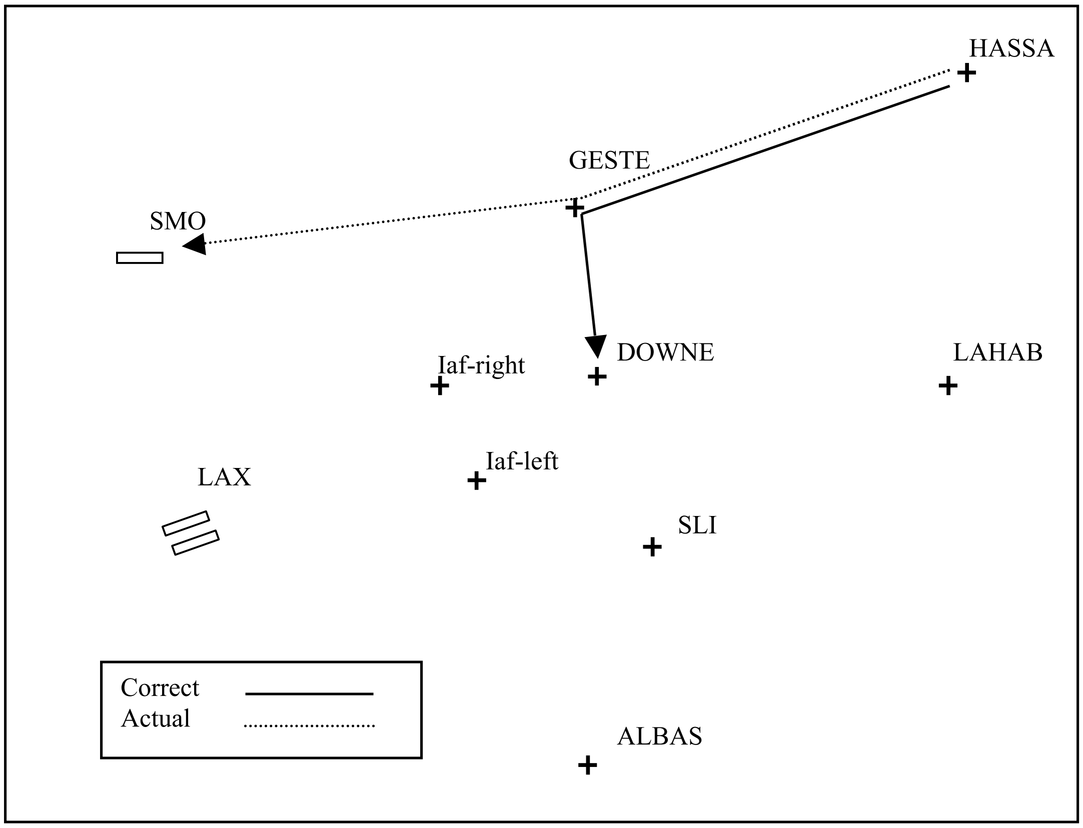
A third method for preventing a predicted error is to train agents to use different, more effective decision strategies. In this case, one could train the controller to always verify runway availability rather than rely on a the high-frequency assumption. Similarly, one could train the controller to engage in self-reminding behavior (mental rehearsal) when a runway closure is indicated, thus increasing the chance that reliance on the assumption will be suppressed.

A fourth and final method is to enhance the effectiveness of a decision strategy by providing reliable perceptual indicators. In this example, the agent had come to rely on an incidental indicator for runway closure (runway imbalance). A designer could modify the radar display to represent runway closures directly. Note that even this method can be problematic. Radar displays must be designed not only to present useful information, but also to avoid visual clutter that might make using the information difficult. Adding a runway closure indicator might therefore solve one problem only to introduce or aggravate another. By explaining why such errors might occur, the model implies a possible solution. In particular, the model explains the error as an indirect consequence of low-workload. One solution then is to display runway closure information in low-workload conditions when incidental reminders are likely to be absent and display clutter is not much of a problem. In high workload when clutter must be avoided, there is no longer a need to display closure information explicitly since runway imbalances will be perceptually apparent, reminding the agent to perform correctly.

7.2 Incident 2 – wrong heading

7.2.1 Scenario

Normally, a plane arriving in LAX TRACON airspace near the HASSA fix (see map below) and bound for landing at LAX airport, will be sent directly from HASSA to DOWNE where it can be set up for a landing approach. However, when airspace near DOWNE has become especially crowded, the controller may choose to divert the aircraft in order to delay its arrival at DOWNE, thus allowing time for crowded conditions to abate. For example, the controller might choose to divert an aircraft to the GESTE, intending to route it from there to DOWNE.



Vectoring a plane from GESTE to DOWNE is unusual. Normally, a plane nearing GESTE is on its way to a landing at Santa Monica airport (SMO). In high workload, high time-pressure conditions, a controller may become distracted and thoughtlessly route a plane nearing GESTE directly to SMO. Though correct in most cases, this action would constitute an error if applied to an LAX-bound aircraft.

7.2.2 Simulation

As the newly arrived aircraft nears HASSA, the controller executes a procedure (not shown) to determine the plane's destination, select a heading based on that destination and current airspace conditions, and then issue a clearance directing the plane to the selected heading. In this case, the controller determines that the aircraft's destination is LAX but, because of heavy air traffic between LAHAB and DOWNE, decides to clear the plane for an indirect (delaying) route via GESTE rather than direct to DOWNE.

Eventually, the plane approaches GESTE, triggering a pending task to handle planes arriving at this waypoint. Executing this task involves carrying out the following procedure.

```
(procedure
 (index (handle-waypoint geste ?p))
 (step s1 (retrieve (cleared ?p ?callsign from geste direct-to ?dest)))
 (step s2 (select-next-waypoint geste ?p => ?next)
      (waitfor (terminate ?s1 failure)))
 (step s3 (clear ?p from geste direct-to ?next) (waitfor ?s2))
 (step s4 (terminate)
      (waitfor (terminate ?s1 success))
      (waitfor ?s3)))
```

After confirming that the plane has not already been cleared from GESTE to another waypoint, the agent selects the next waypoint (either SMO or DOWNE) using the following procedure.

```
(procedure
  (index (select-next-waypoint geste ?plane))
  (step s1 (determine-destination geste ?plane => ?dest))
  (step s2 (compute-best-next-waypoint from geste ?dest => ?best) (waitfor ?s1))
  (step s3 (terminate >> ?best) (waitfor ?s2)))
```

The plane's ultimate destination, SMO or LAX, determines which waypoint it should be directed towards. The agent determines destination using either of two specification methods. The first involves retrieving the information from memory while concurrently finding and reading the destination off the aircraft's flight strip; this succeeds when either activity yields the needed information. The second method relies on frequency bias which specifies that planes located near the GESTE fix are most frequently headed to SMO. In general this value is very likely to prove correct; however, it is not reliable enough to make using frequency bias the normal specification method. Hence, the simulated controller uses the first method to specify destination in normal conditions, but falls back on frequency bias in high-workload conditions when saving effort is especially worthwhile. This strategy for determining destination is represented by the following procedures:

```
(procedure
  (index (determine-destination ?plane))
  (step s1 (select-specification-method det-dest))
  (step s2 (execute-specification-method det-dest ?meth ?plane => ?result)
       (waitfor ?s1))
  (step s3 (terminate >> ?result) (waitfor ?s2))


(special-procedure
  (index (select-specification-method det-dest))
  (if  (equal ?subjective-workload ‘high) ‘read-flightstrip ‘frequency-bias))

(procedure
  (index (execute-specification-method det-dest read-flightstrip ?plane))
  (step s1 (retrieve (destination ?plane ?destination)))
  (step s2 (locate-flightstrip ?plane => ?strip))
  (step s3 (read-word 3 ?strip => ?destination) (waitfor ?s2))
  (step s4 (terminate >> ?destination)
      (waitfor ?s3) (waitfor (destination ?plane ?destination))))

(special-procedure
  (index (execute-specification-method det-dest ‘frequency-bias))
  ‘smo)
```

In normal conditions, the agent will decide where to direct the plane on the basis of reliable destination information. However, in high workload conditions, destination will be determined on the basis of an incompletely reliable heuristic: if a plane is near GESTE, assume it is headed to SMO. While normally correct, reliance on this assumption makes the agent vulnerable. The simulated will thus tend to err in a predictable way in predictable circumstances – i.e. in high workload conditions, it will tend to inappropriately route an LAX-bound plane near GESTE to SMO.

7.3 Incident 3 – wrong equipment

The final scenario to be described in this chapter illustrates how APEX can be used to draw attention to potential usability problems in a still-evolving design. In particular, the scenario describes an incident in which design characteristics of an information display interact with a new communication technology called Data Link to inadvertently facilitate air traffic controller error.

> "One of the primary information transfer problems that constrains the capacity of the current ATC system is the inherently limited communication channel that exists between the air traffic controller and the aircraft pilot. Because this voice radio link operates in a broadcast mode between a single controller and all aircraft operating in the current airspace under his control, frequency congestion is a common occurrence when the volume and complexity of air traffic increases. Such saturation of the communications channel affects the performance of the ATC system by preventing the timely issuance of clearances and by restricting the vital exchange of information upon which safe and efficient operation of the NAS (National Aerospace System) depend.
>
> …Data Link is a digital communications technology which is being developed as a supplement to traditional voice radio for ATC communications. …Data Link communications are distinguished from traditional voice radio in two essential ways. First, unlike analogue voice messages, Data Link messages consist of digitally encoded information. Thus data may be entered for transmission either manually, or by direct access to information contained in airborne or ground-based computers. Furthermore, the capability of a digital system to provide automatic error checking of sent and received messages makes Data Link a highly reliable system which is not susceptible to degradation by interfering noise sources.
>
> The second way in which Data Link differs from the voice radio channel is its capability to discretely address individual receivers. Unlike the simplex radio system which permits only a single speaker to transmit on the broadcast frequency at any point in time, Data Link messages can be sent selectively, and transmission rates are not artificially bounded by the effective speaking and listening rates of the user. As a result, Data Link channels can have a much higher capacity than voice channels and critical messages sent by a controller are assured of receipt only by the intended aircraft. [Talotta, 1992]"

The use of Data Link is limited by at least two factors. First, Data Link clearances take longer to formulate (using keyboard and/or mouse) and longer to receive (read off a display) than voice clearances. They will therefore be inappropriate when fast response times are needed. Second, equipment needed to receive Data Link clearances will not be aboard all aircraft, especially small private planes; and this equipment will sometimes malfunction. In these cases, controllers should revert to issuing clearances over radio.

7.3.1 Scenario

Changes to air traffic control equipment may interact with Data Link in unexpected ways. For example, future information displays will almost certainly use color to display data that is currently presented in a less convenient form. One possibility is that an aircraft's size will be indicated by the color of its radar display icon rather than by text on a flightstrip as it is currently. This would have the beneficial effect of speeding up frequent decisions that depend on knowing an aircraft's size. However, it may also have other effects.

For example, the presence or absence of Data Link capabilities will probably be indicated on an aircraft's datablock. But because aircraft of size large or above will almost always have working Data Link equipment, controllers may learn to use aircraft size as a heuristic indicator rather than consult the datablock.[32] Even though this size heuristic may be quite reliable, its use would lead to error on occasions. For instance, the controller might try to issue a Data Link clearance to an aircraft whose equipment has not been upgraded with Data Link capabilities or has malfunctioned.

7.3.2 Simulation

To decide whether to use voice (radio) or keyboard (Data Link) to issue clearances, the model considers only a single factor – whether the aircraft has working Data Link equipment.

[32] The tendency to rely on simple perceptual actions in place of more demanding perceptual or cognitive actions is thought to be a general feature of human cognition [Vera, 1996].

```
(procedure
 (index (select-clearance-output-device ?plane))
 (step s1 (determine-if-datalink-available ?plane => ?datalink))
 (step s2 (compute-best-delivery-device ?datalink => ?best) (waitfor ?s1))
 (step s3 (terminate >> ?best) (waitfor ?s2)))

(special-procedure
 (index (compute-best-delivery-device ?plane-has-datalink))
 (if (and ?plane-has-datalink) 'keyboard 'radio))
```

This can be determined using either of two methods. First, Data Link availability can be inferred from the color of the plane's radar display icon; since icon color information is easily acquired and often already available in visual memory (see 6.1), this inference method tends to be fast and undemanding of limited resources. Second, the information can be read off the plane's datablock while concurrently retrieved from memory. This method is more demanding but also more reliable.

```
(procedure
  (index (determine-if-datalink-available ?plane))
  (step s1 (select-specification-method det-datalink => ?meth))
  (step s2 (execute-specification-method det-datalink ?meth  ?plane => ?dlink)
(waitfor ?s1))
  (step s3 (terminate >> ?dlink) (waitfor ?s2)))

(procedure
  (index (execute-specification-method det-datalink infer))
  (step s1 (determine-color plane-icon ?p => ?color)
(waitfor (terminate ?s1 failure)))
  (step s2 (infer datalink ?color => ?truthval) (waitfor ?s1))
  (step s3 (terminate >> ?truthval) (waitfor ?s2)))

(procedure
  (index (execute-specification-method det-datalink read ?plane))
  (step s1 (retrieve (equipped ?plane datalink ?truthval)))
  (step s2 (locate-datablock ?plane => ?db))
  (step s3 (shift-gaze ?db) (waitfor ?s2))
  (step s4 (read-datablock-item datalink) (waitfor ?s3))
  (step s5 (encode (equipped ?plane datalink ?truthval))
      (waitfor (textblock 3 ?truthval)))
   (step s6 (terminate >> ?truthval) (waitfor ?s5)))
```

The simulated agent decides between the more and less resource-demanding methods by following a simple rule: always use the less demanding method unless its use has been suppressed. Suppression results whenever the agent detects a condition that would cause the inference method to fail – i.e. a large or heavy plane with non-functioning or non-existent Data Link equipment. This strategy is represented by the following procedure[33]:

```
(special-procedure
  (index (select-specification-method det-datalink))
  (assume ?functioning (functions datalink ?plane true) (15 minutes))
  (assume ?equipped (normally-equipped ?plane true) (15 minutes))
  (if (and ?functioning ?equipped) ‘infer ‘read))
```

The success of this strategy depends on how reliably exception conditions are detected. For instance, if the simulated controller's routine for scanning the radar display entailed reading (and rereading) aircraft datablocks, the absence of otherwise expected Data Link equipment would be detected with great reliability. Alternately, the agent could make sure to read all information contained in a datablock whenever that datablock is consulted for any reason. This more opportunistic method provides a lesser degree of reliability, though possibly enough to keep error rate very low.

The approach used in the model balances the need to maintain an awareness of exception conditions with the need to manage time- and resource-demanding efforts such as reading. In low or medium workload, the simulated controller reads all information contained in the datablock whenever any datablock information item is needed. In high workload, only the item that triggered examination of the datablock is read. Thus:

[33] The model reverts to the inference method 15 minutes after detecting an exception condition indicating that inference will be unreliable. This time value was selected based on the amount of time typically required for an aircraft to pass through a single controller's airspace.

```
(procedure
  (index (read-datablock-item ?item))
  (step s1 (decide read-all-or-none => ?all))
  (step s2 (read-whole-datablock) (waitfor ?s1 :and (equal ?all ‘true)))
  (step s3 (read-single-datablock-item ?item)
     (waitfor ?s1 :and (equal ?all ‘false)))
  (step s4 (terminate) (waitfor ?s2) (waitfor ?s3)))
```

The overall effect of the datablock reading strategy and the clearance-device selection strategy is to produce errors in predictable circumstances. In particular, the simulated controller will tend to inappropriately issue keyboard clearances following a sustained period of high workload in which the unusual absence of a functioning Data Link to a large or heavy aircraft has gone unnoticed.

Assuming that controllers will be properly alerted when Data Link clearances have not or cannot be received, individual errors of this sort will have little effect other than to waste a controller’s time. However, if such errors occur frequently they are likely to reduce productivity and inhibit user acceptance of the new technology. Detecting the problem early in the design process not only makes it possible to avoid these consequences. It also allows designers to consider how best to support and integrate the process of issuing Data Link clearances before expensive commitments have been made to bad equipment and procedure designs.

Chapter 8

Towards a Practical Human-System Design Tool

In principle, computer simulation could provide invaluable assistance in designing human-machine systems, just as it does in designing inanimate systems such as engines and electronic circuits. However, a great deal remains to be accomplished before this approach can become a practical part of the human-machine system design process. This document has focused mainly on the problem of constructing an appropriate human operator model – i.e. one that has the capability to simulate expert-level performance in complex, dynamic task environments, and can be used to predict aspects of human performance, operator error in particular, that are especially relevant to design. Numerous other problems must also be addressed.

1. Human operator model has adequate task-performance capabilities
2. Model can predict design-relevant aspects of human performance
3. Concrete methodology exists for preparing and using the model
4. There is a way to judge when simulation modeling economical
5. No more than moderate effort required to carry out task analysis
6. No more than moderate effort required to construct a simulated world
7. No more than moderate effort required to analyze simulation results

To varying extents, discussion of all of the issues listed above has appeared in previous chapters. This chapter ties together and extends these discussions in order to clarify what has been and what remains to be accomplished to allow human-system simulation to achieve its potential. Section 8.1 summarizes lessons learned in constructing APEX human operator model. These lessons are presented as a set of principles for constructing a human model with appropriate task performance and predictive capabilities (issues 1 and 2 above) given the overarching goal of producing a practical design tool. Subsequent sections will discuss the remaining issues 4-7; the third issue, the availability of a concrete methodology was the main topic of chapter 2 and will not be reviewed.

8.1 Lessons learned in building APEX

The intention to apply a model to help analyze designs should strongly constrain how the model is constructed.

The basic requirements for APEX were (1) that it could model the performance of diverse tasks in complex task environments such as air traffic control, and (2) that its performance could vary in human-like ways depending on the designed elements of its task environment – in particular, that it show approximately human tendency to err. Model-building efforts were driven in part by careful analysis but also in part by trial-and-error. As patterns emerged regarding what would work and what would not, it became possible to infer a set of general guidelines to help direct model-building efforts more effectively. In most cases, these guidelines made a great deal of sense in hindsight, but were not at all obvious at the outset. The approach that eventually emerged can be summarized as a set of six principles, each discussed below in some detail.

1. Make the initial model too powerful rather than too weak.
2. Extend or refine the model only as required.
3. Model resource limitations and coping mechanisms together.
4. Use stipulation in a principled way.
5. Assume that behavior adapts rationally to the task environment.
6. Parameters that are of particular interest may be set to exaggerated values.

8.1.1 Make the initial model too powerful rather than too weak

Human performance models are often evaluated by comparing their behavior to laboratory experimental data. For example, delays in responding to a stimulus in dual-task conditions exhibited by the CSS architecture [Remington et al., 1990] closely approximate human delays in similar conditions. The high degree of fit between human and model performance is meant to provide evidence for the soundness and veridicality of these models. For these kinds of models, the need to characterize the details of human response-time distributions in simple, time-pressured tasks is of paramount importance.

In models intended for practical applications, the detailed accuracy of predicted response-time distributions must be weighed against the sometimes conflicting requirement that the model operate in a complex, multitasking domain. This conflict between accuracy and capability arises from limits on scientific understanding of high-level cognitive tasks such as planning, task switching, and decision-making. To incorporate these capabilities into a model requires extensive speculation about how humans carry out such tasks, supplemented with knowledge-engineering in the domain of interest.

For models meant to be evaluated on the degree to which their performance fits empirical data, a reluctance to incorporate capable but speculative model elements is easily understood. The goal of predicting performance in complex domains prescribes the opposite bias: if human operators exhibit some capability in carrying out a task, the model must also have that capability as a prerequisite to predicting performance. One consequence of this bias is that the model may tend to be overly optimistic about human performance in some instances; the model performs effectively in situations where humans would fail. The APEX approach is based on the view that the model's need to operate in interesting domains (where the need to predict design-facilitated error is greatest) outweighs the resulting reduction in detailed accuracy.

8.1.2 Extend or refine the model only as required

Early versions of the current model were developed with the idea that any reliable psychological finding that could be incorporated into the model constituted a useful addition. The initial goal was thus to bring together as much psychology, psychophysics, neuroscience, anthropometry, and so on as possible. However, it sometimes happened that findings incorporated as elements of the model added insufficient value to compensate for difficulties they created.

For example, early versions of the model incorporated the finding that human vision takes slightly longer to process certain perceptual features than others; color, for instance, takes a few milliseconds longer to process than orientation or primitive shape. Of the kinds of predictions the model could reliably make or were in prospect, none depended on this aspect of the model. Moreover, its inclusion was quite costly since it forced simulation mechanisms to consider very brief time intervals (one millisecond), thus slowing simulations substantially.

Adding unnecessary detail to the model makes it slower in simulation, increases the amount of effort needed to make future improvements, and makes it harder to debug, explain, and evaluate. Therefore, it is important to make sure that extensions to the model make it more useful. For current purposes, that means any added feature should help to highlight opportunities for operationally significant human error.

Another instructive example arose in modeling time delays that occur when an agent tries to acquire information for a decision task. For example, a controller deciding which runway to direct a plane towards must acquire information on such factors as the relative number of planes lined up for each alternative runway, the weight of the plane (heavy planes should preferably be sent to the longer runway), and whether each runway is operational. To acquire information about any of these factors from memory, a controller would have to employ his/her MEMORY resource which can only be used for one retrieval task at a time [Carrier and Pashler, 1995].

Since use of the MEMORY resource blocks its availability to other decision-making tasks (and also delays the current decision task), the amount of time required to perform a retrieval can be an important determiner of overall performance. Incorporating the determinants of retrieval time into the model would thus seem to have great value in predicting performance. However, two other factors suggest the need for care in deciding what aspects of memory retrieval should be modeled. First, a survey of the literature on memory reveals numerous factors affecting retrieval time. Incorporating each of these factors into the model would likely involve an enormous commitment of time and effort.

Second, controllers typically have alternative ways to evaluate the factors that bear on their decisions, each varying in required time and other properties. For example, to acquire information about the weight class of a plane, a controller can (a) read the weight value off the plane’s data block on the radar display, (b) retrieve that plane’s weight from memory, or (c) assume that the plane has the same weight class as most other planes. The time required to carry out these methods can differ by orders of magnitude. In the current model, relying on a default assumption requires no time or resources; memory retrieval requires approximate .5 seconds; visual search and reading require a highly variable amount of time ranging from 0.5 seconds to 10 seconds or more. The magnitude of these differences imply that refinements to the model that increase its ability to predict which information acquisition method will be used are generally more valuable than refinements that account for variance in memory retrieval time.

These experiences imply two corollaries to the principle of letting modeling goals drive refinement efforts. First, as illustrated by the example of modeling differential propagation rates of low-level visual features, one should prefer to maximize the temporal coarseness of the model with respect to the desired classes of predictions. Elements that rely on temporally fine-grained activities should be included only if their inclusion accounts for significant differences in overall task performance. Second, as illustrated by the memory modeling example, prefer to model the largest sources of performance variability in a given activity before modeling smaller sources.

8.1.3 Model resource limitations and coping mechanisms together

People's tendency to err is often explained as a consequence of limitations on perceptual, cognitive, and motor resources (e.g. [Reason, 1990], [Kitajima and Polson, 1995], [Byrne and Bovair, 1997]). However, the most obvious ways of linking errors to resource limitations may be misleading. In particular, each limitation can be associated with a set of behaviors used to cope with that limit. These coping behaviors rely on assumed regularities in the world and on other assumptions that can sometimes prove false. The imperfect reliability of a coping method's underlying assumptions renders people susceptible to error. This is something of a reconceptualization of the problem, as it moves the problem locus from peripheral resources which are somehow “overrun” by task demands, to learned strategies employed by the model’s plan execution component.

For example, people cope with a restricted field of view by periodically scanning their environment. Mechanisms for guiding the scan must guess where the most interesting place to look lies at any given time. By making some assumptions, for example, that certain conditions will persist for a while after they are observed, scanning mechanisms can perform well much of the time. But even reliable assumptions are sometimes wrong. People will look in the wrong place, fail to observe something important, and make an error as a result. Of course, people have no choice about whether to scan or not; if a person were somehow prevented from scanning, many tasks would be impossible. By forcing people to guess where to look, a limited field of view enables error.

Human resource limits are much easier to identify and represent in a model than are the subtle and varied strategies people use to cope with those limits. For example, people have limited ability to ensure that the things they have to remember “come to mind” at the right time. Modeling this requires the separation of the processes that determine the result of a retrieval attempt from those that initiate a memory retrieval attempt. Retrieval initiation happens only when triggered by certain conditions external to the memory model itself.

People cope with memory limitations by maintenance rehearsal, writing notes to themselves, setting alarms, and other methods. Unless the model includes mechanisms needed to carry out these strategies, it will tend to under-predict human performance – that is, it will predict failures of memory where people would not actually fail. As discussed above, present purposes require that when an accurate model is not possible, exaggerating human capabilities should be preferred to understating them. Applied to the problem of accounting for innate human limitations, this strongly suggests an emphasis on representing coping strategies for circumventing any modeled limitation.

8.1.4 Use stipulation in a principled way

While it is challenging and scientifically worthwhile to show how intelligent behavior can emerge from the harmonious interplay of myriad low-level components, practical considerations require that these low-level component processes be modeled abstractly. In some cases, the need for abstract process models arises from the practical considerations already discussed -- that is, to avoid complicating the model with elements that add little to its power to make useful predictions. In other cases, scientific ignorance about how processes are carried out requires a model to stipulate that a process occurs without specifying any mechanism.

For example, simulating the behavior of air traffic controller (and operators in most other domains of practical interest) requires model elements representing human vision. Construction of these elements had to proceed despite the fact that no complete and detailed model of human visual processing currently exists. In fact, no existing model of visual processing, including robot vision systems designed without the requirement that they conform to human methods or limitations, can achieve anything close to human visual performance. Thus, the mechanism of normal visual function could not have been represented, even if doing so would have been worthwhile in terms of previously described goals.

Instead, the model requires that the simulated controller operate in a perceptually simplified world in which a detailed representation of the visual scene, for example, as an array of intensity- and chroma-valued pixels, is abandoned in favor of qualitative propositions representing properties of visible objects. For instance, to represent planes observable on a radar display, the world model generates propositions such as

(shape visual-obj-27 plane)

(color visual-obj-27 green)
(location visual-obj-27 (135 68))

which together represent a green airplane icon located at a given position relative to a reference point.

To simulate nominal visual performance, the vision model simply passes propositions from the world to cognitive model elements. Thus, decision-making elements would, in some cases, simply be informed that there is an object at location (135,68) without vision having to derive this information from any more fundamental representation.

Given the goal of accounting for the effect of interface attributes on controllers' performance, the need to eliminate any explicit representation of visual processing poses an important problem: How can the effect of interface attributes such as color, icon shape, and the spatial arrangement of visual objects be accounted for except by allowing them to affect processing? To illustrate the present approach, consider how the model handles direction of gaze, one of the most important determinants of what visual information is accessible at a given moment.

The first step was to construct a basic model of visual processing that would successfully observe every detail of the visual environment at all times. As described, this simply required a mechanism that would pass propositions describing the visual scene from the world model to cognitive mechanisms. Real human visual performance is, of course, limited to observing objects in one's field of view. Moreover, the discriminability of object features declines with an object's angular distance from fixation (the center of gaze). To model this, the model requires propositions generated by the world include information on the discriminability of the visual feature each represents. For example, the proposition

(shape visual-obj-27 plane)

means that *visual-object-27* can be recognized as an *plane* as long as detectability requirements imposed by the VISION resource are met.

This is an unusual way of looking at the process of acquiring information. Rather than modeling the process of constructing information from perceptual building-blocks, all potentially relevant information items are considered potentially available; the model simply stipulates that the constructive processes operate successfully. The task of the model then is to determine which potentially available items are actually available in a given situation.

A generalization of this approach is also used in non-perceptual components of the model. In general, nominal performance is stipulated, and factors that produce deviations form nominal performance are modeled separately and allowed to modulate nominal performance.

8.1.5 Assume that behavior adapts rationally to the environment

For many performance variables of interest, the amount of practice constitutes the single largest source of variability. People become adapted to their task environment over time, gradually becoming more effective in a number of ways [Anderson, 1990, Ericsson and Smith, 1991]. For the purpose of modeling highly skilled agents such as air traffic controllers, this process has several important consequences.

First, people will, over time, come to learn about and rely on stable attributes of the task environment. For instance, in the air traffic control scenario discussed in section 2.2, the controller relied on the (false) default assumption that both runways were available. Constructing a model to predict such an error thus requires determining that certain conditions are much more common than others and are likely to be treated as default assumptions by experienced operators. Similarly, the controller in this example relied on a perceptual cue, a linear arrangement of plane icons on the radar display, to signal that a non-default condition might hold and that a memory retrieval action was warranted. Thus the model requires determining what kinds of perceptual cues are likely to be available in the environment and to be exploited by experienced operators to support cognition.

A second consequence of adaptation that should be considered in the construction of models such as APEX is the fact that, over time, people will learn which policies and methods work and which tend to fail. This significantly complicates analyses of the effect of innate human resource limitations on task performance. For example, experienced grocery shoppers will come to learn that relying on memory to retain the list of desired goods does not tend to work very well. Experienced shoppers will almost inevitably come to rely on some strategy that circumvents the limitations on their memory [Salthouse, 1991]. For example, some will rely on a written list; others might learn to scan the shelves for needed items, thus replacing a difficult memory task (recall) with an easier one (recognition).

To account for the effect of limitation-circumventing strategies, the APEX Procedure Definition Language includes a variety of notations for representing behaviors that incorporate these strategies. For instance, procedures representable in the model can carry out visual search tasks that result in the initiation of a memory retrieval, thus supporting decision-making and prospective memory tasks that depend on timely memory retrieval. However, the ability to represent such procedures must be coupled with some method for determining what strategies experienced practitioners will tend to employ, and thus what procedures should be represented. This issue is discussed in section 2.4.2.

The assumption that experienced practitioners will have adapted to their task environment provides a basis for setting otherwise free parameters in the model. For example, the model's prospective memory, a key element of the model for predicting habit capture errors, assumes that the likelihood that a person will attempt to verify a default assumption by retrieving information from memory declines over time. For example, the air traffic controller in the example scenario became less likely to retrieve knowledge about the runway closure from memory as time elapsed since the last time s/he was reminded of the closure by a visible anomaly on the radar display.

Constructing the model required making an assumption about the rate at which retrieval likelihood would decline. Note that this value could, in principle, be obtained in a controlled experiment. However, performing such an experiment would undermine the whole purpose of the modeling effort which is to provide performance estimates in the absence of empirical testing. The approach used was to assume that the retrieval-attempt likelihood function depended only on considerations of utility, and not on any innate limitations.

In particular, three factors were considered. First, air traffic controllers must generally learn to minimize the use of limited cognitive resources in decision-making to cope with potentially very high workload. Thus, optimal decision-making performance must avoid memory retrieval whenever the result is likely to confirm a
default assumption. Second, regularities in the duration of a given non-default condition indicate that, after a certain interval, decision-mechanisms can once again reliably assume the default.

Third, regularities in the rate at which perceptual indicators of the non-default condition are observed can provide an accurate determination of when the default condition has resumed -- that is, if an indicator is usually observed within a given interval, the absence of that interval for the interval can be treated as evidence for the default. Thus,

Memory-retrieval-likelihood = min[D(p),I(p)]

where D(p) is the maximum duration of the non-default interval with likelihood p, and I(p) is the maximum interval between successive observations of a non-default indicator with likelihood p.

Functional estimates of this sort are famous for producing bad theories in certain areas of science such as evolutionary biology. But in the absence of extensive empirical research, the assumption that parameters will have been set by some optimizing adaptive process [Anderson, 1990] will often be a good approximation and will usually constitute the most conservative available guess.

8.1.6 Parameters of particular interest may be set to exaggerated values.

A primary purpose of the APEX model is to highlight vulnerability to human error in complex, dynamic domains. Like many other domains where predicting design-facilitated error would be useful, operating in air traffic control requires a powerful (highly capable) model of how actions are selected. Furthermore, the air traffic control system is operated by highly skilled individuals, and the system itself is designed to prevent or manage errors with extremely high success rates.

For practical purposes, it is not particularly useful to simulate the actual (almost negligible) error rates of the existing air traffic control system. Therefore, having built a capable model and selectively introduced constraints and coping mechanisms, an APEX user should be able to choose parameter values that exaggerate the simulated operator's vulnerability to error. For example, the model may be parameterized with unrealistically pessimistic assumptions about working memory capacity, in order to exaggerate the dependence upon perceptual sources of information. Lewis and Polk [1994] used this technique to model an aviation scenario in SOAR in such a way as to highlight the need for perceptual support: they used a SOAR model with a zero-capacity working memory.

The need for this bias stems from the fact that a designer is usually interested in counteracting even low probability errors, especially when the consequences of error are high or where the task will be repeated often. If low probability errors only showed up in simulation with low probability, the model would often fail to draw attention to important design flaws.

8.2 The economics of human-system modeling

Human-system modeling can entail substantial costs since preparing and using such a model, a process described in chapter two, is likely to require a great deal of time and effort from highly skilled practitioners. A mature approach to modeling should thus include some way to assess whether the overall benefit of carrying out a modeling effort is likely to outweigh the costs. Several issues need to be considered in making this determination, including:

1. Are design improvements likely to have economically significant consequences?
2. Can modeling be used to identify relevant improvements?
3. Is modeling the best approach for doing so?

The potential economic significance of an improved design depends on how the system is to be used. For example, improvements that reduce operator error rate will tend to be significant for safety-critical applications such air traffic control displays; in other applications, occasional operator error may have little importance. Similarly, improvements that enhance operator productivity – for example, by allowing an often-repeated task to proceed more quickly – will be important for some applications but not others. Roughly speaking, there are four kinds of design improvements to be sought from modeling: 1) improvements that reduce the likelihood of a rare, very undesirable event (e.g. potentially catastrophic error) by a small amount; 2) improvements that reduce the likelihood of a common, somewhat undesirable event by a large amount; 3) improvements that reduce the cost (e.g. time-cost) of a rare activity by a large amount; and 4) improvements that reduce the cost of a commonplace activity by even a small amount.

Knowing how a design might benefit from model-based evaluation is helpful in that is allows a modeler to make sensible decisions about which aspects of human performance to model in detail and which to model crudely. It also makes it possible to determine whether a given modeling framework is likely to provide adequate predictive capabilities. APEX, as described, has some ability to predict certain kinds of operator error. Other frameworks, notably GOMS-MHP [Card et al., 1983; Gray et al., 1993], focus on predicting time requirements for routine tasks. The decision whether to engage in a model-building effort should depend, in part, on whether the predictive capabilities of any existing modeling framework can be matched to the prediction needs of the modeler.

As discussed in chapter 2, modeling is most likely to be helpful at a very early stage in design when little time and effort has been committed to the existing design, changing the design can be accomplished at relatively low cost, and traditional user testing cannot yet be carried out. However, simulation modeling is not the only way to assess at an early stage. Other methods, particularly guideline-based methods [Smith and Mosier, 1986] and cognitive walkthroughs [Polson et al., 1992] will sometimes be more appropriate. The relative merits of these approaches are considered in section 2.1.

8.3 Minimizing the cost of modeling

The use of human-simulation modeling as a practical design tool depends not only on the existence of an appropriate human operator model, but also on whether the cost of carrying out a simulation study can be justified. In particular, task analysis, constructing a simulated task environment, and analyzing the results of a simulation can pose prohibitive costs if nothing is done to contain them. The APEX approach has begun to incorporate cost-saving measures, although much has been left to future work.

8.3.1 Task analysis

The term **task analysis** [Mentemerlo and Eddowes, 1978; Kirwan and Ainsworth, 1992] refers to the process of identifying and then formally representing the behavior of a human operator. Even in the best of circumstances, identifying important regularities in operator behavior poses difficulties. For example, while phraseology and some other aspects of air traffic controller behavior can be determined by reference to published procedures (e.g.[Mills and Archibald, 1990]), behaviors such as maintaining an awareness of current airspace conditions are not described in detail by any written procedures. These aspects of task analysis require inferring task representation from domain attributes and general assumptions about adaptive human learning processes.

Prospects for human-simulation modeling as a practically useful design tool depend on developing more effective techniques. Observation-based methods are improving [Kirwan and Ainsworth, 1992]. However, a great deal of work needs to be done to enable task analyses for behaviors which cannot be easily observed because they are rare, covert, or relate to procedures and equipment which have not yet been implemented. Rational Analysis provides a particularly promising approach (see [Anderson, 1990] and section 6.2.3 of this document).

APEX facilitates task analysis in several ways. First, it includes a notational formalism called PDL which anticipates many common elements of a task analysis. For example, the PDL construct PERIOD is used to describe how a person manages repetitive tasks such as checking the speedometer in an automobile or watering one’s plants. Syntactic elements that provide a natural way to think about human behavior make easy to represent cognitive processes underlying behavior. Second, sections 3.4 and 4.6 describe techniques for using PDL to represent common behaviors such as recovering from task failure by trying again, resuming after an interruption by starting over, and ending a task when either of two subtasks has completed successfully. These techniques are to task analysis what “textbook” algorithms and “programming idioms” are to computer programming. The problem of task analysis may be simplified once a number of such techniques have been identified and catalogued.

Third, APEX facilitates task analysis using general purpose procedure libraries. The **innate skills library** includes procedures that describe completely domain-independent human behaviors such as how to sequentially search a visual display for some target object, and the tendency to visually examine newly appearing visual objects (“abrupt onsets”). The **general skills library** includes procedures for such commonplace activities as typing a word on a keyboard, using a mouse to drag an icon to a new location, and reading information from a known location (e.g. a page number in a book, an aircraft’s altitude on its datablock). With a variety of common behaviors represented, such libraries could greatly reduce the effort required to perform a task analysis. Currently, the APEX procedure libraries contain only a small number of procedures; enhancements to this element of this system are considered an important area for future work.

8.3.2 Constructing the simulated world

Preparing a human-system simulation requires not only creating an appropriate operator model, but also modeling the inanimate elements of the operator’s environment – the simulated world. Ideally, most of the effort required to construct a new simulated world would relate to unique aspects of the task domain. For example, in constructing a simulated flightdeck (cockpit) in order to model the performance of the aircraft’s pilot, the modeler should not have to specify the general characteristics of common control elements (e.g. buttons, switches, dials, keyboards) and display elements (e.g. analog meters, textblocks, mouse-selectable icons). Currently, APEX includes only a few, simple models of generic display elements (icons, textblocks) and input devices (mouse, keyboard). A great deal of work in this area remains to be accomplished.

8.3.3 Analyzing simulation output

Human-system simulation provides an approximation of the evaluation data that designers would receive after testing with live users. Consequently, this approach suffers from one of the main difficulties associated with user testing – having to somehow make sense of vast quantities of data. For simulation modeling, this data takes the form of a **simulation trace**, a record of all events that occurred over the course of a simulation scenario. APEX currently provides a partial solution in the form of mechanisms for controlling which events will appear in the simulation trace given to an APEX user. This allows the modeler to focus on events of probable interest and to obtain an account of a scenario at the desired level of detail.

While useful, these mechanisms only begin to address the problem. A full simulation trace may contain dozens of events for each simulated second. Even if a large fraction are suppressed, the number of events occurring over a typical 3 or 4 hour scenario could still be quite large. Additionally, certain kinds of events will be unimportant most of the time, but will occasionally be crucial in accounting for operator error and other significant events. With only existing trace control mechanisms, a modeler must choose either to suppress potentially important detail or cope with it in abundance.

The only solution likely to prove satisfactory in the long run will be to automate the analysis of trace output. Currently, APEX provides no functionality related to the analysis of simulation traces. However, techniques developed by artificial intelligence researchers may prove helpful. In particular, one can view many simulation events of interest, operator error for example, as anomalies that need to be explained using generalized explanation patterns [Schank, 1986]. Such an approach is outlined in section 2.7.2.

8.4 Final Notes

In safety-critical domains, such as nuclear power, aerospace, military, medical, and industrial control systems, the cost and risk of implementing new technology are major barriers to progress. The fear of innovation is not based on superstition, but on the common experience of failure in complex system development projects [Curtis et al., 1988]. Retaining the status quo, however, becomes less and less tenable as existing systems become obsolete and the cost and risk of maintaining them escalate.

Replacement or significant upgrading of such systems eventually becomes inevitable. Therefore, it is necessary to attack the core problem, namely, the lack of a systematic design method for complex human-computer systems. It is the absence of such a methodology that lies at the root of valid concerns about the safety [Leveson, 1995] and economic benefit [Landauer, 1995] of new human-computer systems.

APEX is intended to be a contribution toward improving the design of safety-critical and high-use human-computer systems, for example, the next-generation air traffic control system. The key innovations of APEX are a powerful action selection framework that makes it possible to model performance in complex environments such as air traffic control, a simple, but potentially useful account of certain kinds of human error, and the beginnings of an integrated approach to preparing, running, and analyzing APEX simulations. With additional progress in each of these areas, human simulation modeling may someday become an indispensable tool for developing human-machine systems.

Appendix
APEX Output

This chapter begins with a discussion of how to read and control APEX output. A simple air traffic control scenario is then used to illustrate APEX behavior.

A.1 Reading a simulation trace

The output of an APEX simulation – the **simulation trace** – includes 6 major event types, each associated with a unique designation.

A	Action-selection events
C	Cognitive events
S	Resource signals
R	Resource cycles
W	World events
*	Special signals

Action-selection events in the simulation trace, signified by the letter *A*, represent task processing actions by the action selection architecture; examples include *creating* a task, *testing* a task to see whether all of its preconditions have been satisfied, *executing* an enabled task, and several others[34]. For instance, the following trace item denotes that at time 4220 (4.22 simulated seconds after the beginning of the scenario), preconditions associated with task-1426 were tested and found to be satisfied.

```
[4220-A]  TESTING preconditions for
     [TASK-1426 (DOUBLECLICK-ICON [VOF:{DL603}]) {PENDING}]…. SATISFIED
```

When tasks appear in a trace, three attributes are shown: a unique identifier, a description of the activity prescribed by the task, and its current enablement state (either *pending*, *enabled*, *ongoing*, or *suspended*). Task activity descriptions often refer to visual object files (see section 5.1) denoted by the prefix `VOF`. These are internal representations of visually detected world objects. For example, the object `[VOF:{DL603}]` above refers to an aircraft icon appearing on the radar scope. Use of an aircraft callsign (DL603 is short for Delta flight 603) to identify an aircraft icon is for clarity only; its appearance in a simulation trace does not means that the callsign information has become available to the simulated controller.

As discussed in section 3.1.4, **cognitive events** (denoted by a `C`) are information items that become available to action-selection mechanisms from perceptual resources or from the action selection architecture itself. The most common cognitive events from a perceptual source (VISION) are *new*, *refresh*, and *revalue* propositions (see section 6.1). From action selection mechanisms, the most common cognitive event is a *terminate* event, indicating that a specified task has been terminated.

Resource signal events (`S`), as described in section 3.1.4, are messages from action selection telling a given resource to begin a specified activity. For example, the following event indicates that at time 8200, the VOCAL resource was told to say the phrase "descend and maintain."

```
[8200-S] -->  VOCAL  (DESCEND AND MAINTAIN)
```

[34] Other action selection events: enabling, terminating, … as well as multitask actions (see chapter 5) grabbing (attempting to acquire contested resources), interrupting, and resuming. By convention, all action-selection events end in "-ing."

As will be discussed in chapter 6, each resource in the APEX resource architecture produces action at intervals, specified separately for each resource, called "resource cycles." Activity at each cycle is denoted by a **resource cycle event** (`R`) such as

```
[8700-R]  VOCAL: DESCEND
```

which indicates that during the last cycle, the VOCAL resource uttered the word "descend." Resources that affect the state of the simulated world – e.g. LEFT (the left hand) and VOCAL – can directly indicate the overall meaning of an action sequence to the world simulation software. For instance, the resource signal represented by

```
[8200-S] -->  VOCAL  (DL603 DESCEND AND MAINTAIN 6000)
 {(ALTITUDE DL603 6000)}
```

entails that after the VOCAL resource has finished uttering all of the specified words, it should send the value (ALTITUDE DL603 6000) to the simworld. The simworld can thus determine the meaning of the previous word sequence (that the simulated controller wants the altitude property of plane DL603 to be changed to 6000) without having to spend effort parsing the sequence. These **special signals** are indicated by asterisked trace events such as:

```
[10700-*]  (ALTITUDE DL603 6000)
```

The final type of event appearing in a simulation trace is a **world event** (`W`). World events are occurrences in the simulated world that cannot be directly perceived by the simulated agent, but may be of interest to an APEX user. For example, the following event denotes the arrival of flight DL603 into local airspace where it will usually (but not always) be detected by radar.

```
[4000-W]  (ARRIVAL AC:(DL603c54-180-23-(43.700 37.598)))
```

Representations of aircraft appearing in a simulation trace world event include several pieces of information. The first item following the letters AC (for aircraft) designate the aircraft's callsign. This is followed by either a `c`, `t` , or period character, indicating that the controller currently responsible for the plane is the center controller, the tower controller, or the TRACON controller (self) respectively. Next is a value for the plane's current altitude in hundreds of feet above sea level, followed by a trend character. Trend characters are ^, -, and v, indicating respectively that a value is increasing, holding steady, or decreasing. The next items are a heading value, a trend character for heading, an airspeed value (knots), and then an airspeed trend character. Finally, the plane's location is given in nautical miles East and North of a point corresponding to the Southwest corner of airspace depicted on the radar display.

A.2 Controlling simulation output

A fully detailed APEX simulation trace includes a very large number of events, often several dozen for each simulated second. Trace control mechanisms regulate the amount of detail by filtering out certain event types. Four LISP functions are used to access these mechanisms. The function *show-level* allows a user to set detail in a coarse way. For example,

```
(show-level 1)
```

causes a simulation to include only world-events and special events, suppressing all others. The default show-level, 3, suppresses all action-selection events, all resource cycles, and several common but not usually significant cognitive-events. Complete detail is provided at show-level 6.

The function *unshow* suppresses a specific event type. To suppress resource cycle or resource signal events, the forms *(unshow <resource> cycles)* and
(unshow <resource> signals) are used. For example,

```
(unshow gaze signals)
```

filters out trace events for signals sent to the GAZE resource. Similarly,

```
(unshow all signals)
```

filters out resource signal events for all resource types. The form *(unshow <event-type>)* is used to suppress cognitive or action-selection trace events. For example,

```
(unshow testing)
```

suppresses reporting of tests for satisfaction of task preconditions.

To cause simulation trace mechanisms to begin reporting a previously suppressed event-type, a user employs the function *show*. Show has the same argument structure (and opposite effect) as *unshow*. The final trace control function, *customize-trace*, allows a user to define the currently selected show preferences as a show-level.

```
(show-level 3)
(unshow gaze signals)
(show testing)
(customize-trace alpha)
```

For example, the preceding sequence of trace control commands sets the current trace preferences to a variation on show-level 3 and then defines a new show-level called *alpha*. If at some future time, the command *(show-level alpha)* is given, trace preferences will revert to those that held when *alpha* was established.

A.3 A simple air traffic control scenario

The following simple scenario will be used to illustrate APEX behavior and trace output. Subsequent sections will examine APEX simulation traces of this scenario (edited and annotated), first covering the entire scenario with most detail filtered out, and then covering a few moments in much greater detail.

1. Singapore Air flight 173 (SQ173), arrives in Los Angeles TRACON airspace near the LAHAB fix (see next page). An aircraft icon, datablock, and flightstrip for the new arrival appear at appropriate places on the air traffic control display.
2. The controller detects the new arrival and gives permission to enter TRACON airspace by selecting its icon with a mouse-controlled pointer and double-clicking.
3. The controller determines the plane's destination (LAX) by checking its flightstrip
4. As SQ173 nears the LAHAB fix, the controller verbally clears it to descend to the LAX approach altitude of 1900 feet and then vectors it (clears to a new heading) towards DOWNE.
5. As SQ173 nears DOWNE, the controller decides to land the plane on LAX's left runway and then vectors it towards the left runway initial approach fix (Iaf-Left).
6. As the aircraft nears Iaf-Right, the controller vectors it towards LAX and then hands off control of the plane to the LAX tower controller.

A.3.1 Examining a low-detail, long timeline trace

At the beginning of an APEX simulation run, the system queries the simulated world software for information about the upcoming scenario, and then outputs this information. This will include a list of scenario parameters and scheduled events. The following scenario description consists of a single event: the arrival of an aircraft at time 4006 (4.006 seconds following the start of the simulation).

```
----- Starting simulation run -----

4006 ARRIVAL AC:(SQ173c49-306-26-(50.0 14.76378))
```

The scenario begins with the simulated controller scanning the radar display for aircraft. As described in section 5.3.3, an agent performs a visual scan by sequentially examining spatial regions, thereby learning the approximate number of visual objects contained by the region, and potentially other information as well. In this case, there are no aircraft on the radar display. The agent examines regions 1a, then 4, then 7, and so on, learning in each case that the current object count is zero. Scan-driven gaze shifts and resulting visual situation updates are frequent events in the simulation, occurring approximately every .75 seconds. With some exceptions, these events have been edited out of this trace.

```
[200-S] --> GAZE ((LOCUS 1A)) TASK114082
[550-C] (NEW (COUNT 1A 0))
[1050-S] --> GAZE ((LOCUS 4)) TASK114096
[1300-C] (NEW (COUNT 4 0))
[1800-S] --> GAZE ((LOCUS 7)) TASK114105
[2050-C] (NEW (COUNT 7 0))
[2550-S] --> GAZE ((LOCUS 6)) TASK114114
[2800-C] (NEW (COUNT 6 0))
[3300-S] --> GAZE ((LOCUS 3)) TASK114123
[3550-C] (NEW (COUNT 3 0))
```

The radar display updates every 4 seconds, resulting in this case in an approximately 4 second delay between the aircraft's actual arrival in TRACON airspace (t=4006), and its appearance on the radar display (t=8000). The agent does not actually detect the arrival until some time after a scan-driven gaze-shift to region 4A has completed. Because the agent is not actually gazing (attending) to the newly appearing plane-icon, its visual system can only detect certain kinds of "pre-attentively available" information (see 5.1.2) such as approximate shape and blink rate.

```
[8000-W] (ARRIVAL AC:(SQ173c49-306-26-(50.0 14.76378)))
[8470-S] --> GAZE ((LOCUS 4A)) TASK114202
[8800-C] (NEW (SHAPE [VOF:{SQ173}] (PLANE ICON VISOB)))
[8800-C] (NEW (BLINK [VOF:{SQ173}] 2))
[8800-C] (NEW (COUNT 4A 0))
```

The newly appearing visual object triggers a pending task to examine any "abrupt visual onset." This involves first interrupting the current visual examination task (interruptions will be discussed in depth in chapter 5) and then sending a cognitive signal to the GAZE resource to shift locus to the new object.

```
[8800-A] INTERRUPTING [TASK114065 (MONITOR-GEODISPLAY-REGION 4A)
 {SUSPENDED}]
[8800-S] --> GAZE ((LOCUS [VOF:{SQ173}])) TASK114219
```

After classifying the new object as an arriving plane, the agent decides to accept the plane into its airspace. This "handoff" process requires first double-clicking on the plane icon with a mouse-controlled pointer. The action-selection events underlying this point-and-click behavior will be discussed in detail in the next section. Note that the simulated world display updated airspace information at 4 second intervals. The simulated controller does not have access to this information.

```
[9550-S] --> LEFT ((GRASP MOUSE)) TASK114241
[11050-*] (GRASP MOUSE)
[11150-S] --> LEFT ((MOVE POINTER {SQ173})) TASK114236
[12000-W] (LOC (49.73 14.96) (50.0 14.76378) AC:(SQ173c49-306-26-(49.73
    14.96)))
[12650-*] (MOVE POINTER {SQ173})
[12750-S] --> LEFT ((DOUBLECLICK)) TASK114222
```

Once the double-click action has been initiated, the agent continues the handoff process by finding the aircraft's identifying callsign and encoding that information in memory. Callsign data is located in the first word of the datablock associated with an aircraft icon. Reading (see 5.3.4) this item requires shifting the locus of visual attention to the datablock and then specifying which word (position in sequence) is the feature of interest.

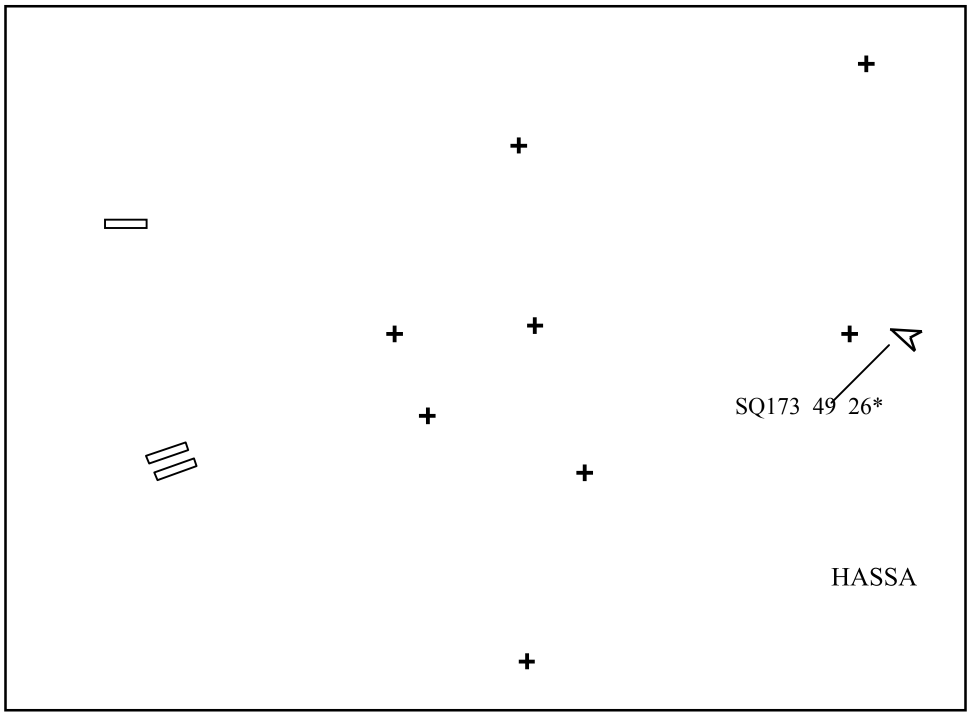
```
[12750-S] --> GAZE ((LOCUS [VOF:{db-114192}])) TASK114260
[13250-S] --> GAZE ((FEATURE 1)) TASK114232
[13550-C] (NEW (TEXT [VOF:{db-114192}] SQ173))
[13550-S] --> MEMORY ((ENCODE (CALLSIGN [VOF:{SQ173}] SQ173))) TASK114217
[14150-C] (NEW (CALLSIGN [VOF:{SQ173}] SQ173))
```

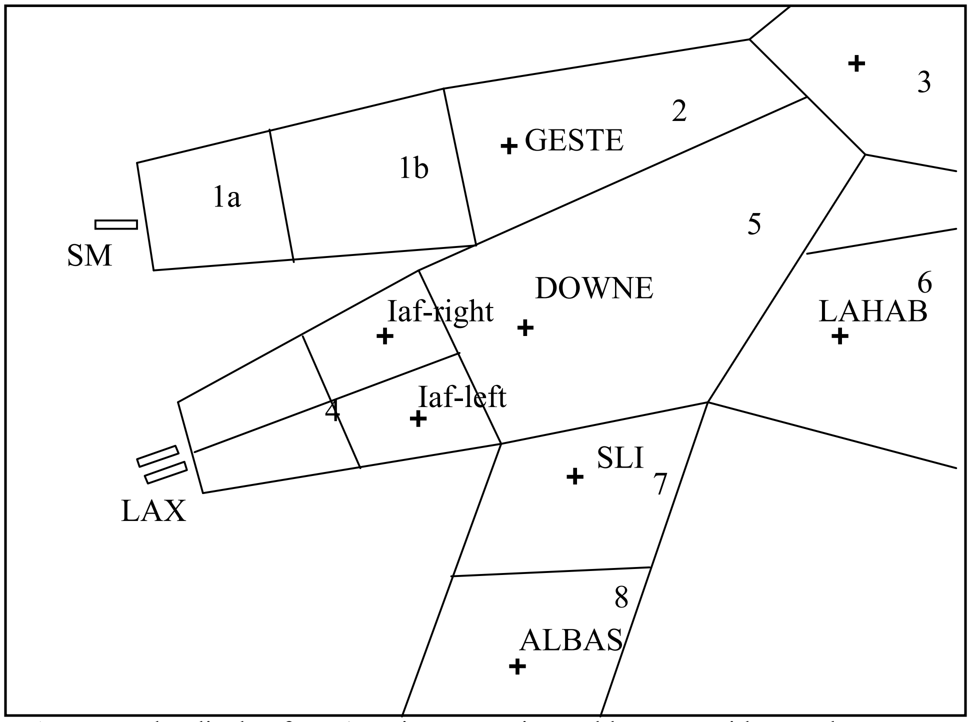
After reading and encoding the callsign, the agent finds the aircraft's flightstrip, reads its destination, and encodes this in memory. During this action sequence, the double-click completes.

```
[14150-S] --> GAZE ((LOCUS [VOF:{strip-114193}])) TASK114280
[14250-*] (DOUBLECLICK)
[14300-C] (REVALUED (BLINK [VOF:{SQ173}] 0))
[14550-S] --> GAZE ((FEATURE 3)) TASK114274
[14800-C] (NEW (TEXT [VOF:{strip-114193}] LAX))
[14800-S] --> MEMORY ((ENCODE (DESTINATION [VOF:{SQ173}] LAX))) TASK114215
[15400-C] (NEW (DESTINATION [VOF:{SQ173}] LAX))
```

Once the process of accepting a handoff has finished, the agent continues scanning the scope. The interrupted task of examining region 4A is resumed, causing it to reset (start over). When the scan gets to region 6, visual processes determine that old information on how many objects are in the region has become obsolete, causing it to become revalued (updated). Finding objects in a radar display region triggers an process to determine which objects are planes and whether any are approaching a position that mandates a response. For example, in region 6, planes near the LAHAB fix must be vectored (rerouted), usually towards DOWNE, and from there to LAX.

```
[15550-A] RESUMING [TASK114065 (MONITOR-GEODISPLAY-REGION 4A) {ONGOING}]
[15550-C] (RESET [TASK114065 (MONITOR-GEODISPLAY-REGION 4A) {ONGOING}])
[15550-S] --> GAZE ((LOCUS 4A)) TASK114297
[16270-S] --> GAZE ((LOCUS 7)) TASK114309
[17050-S] --> GAZE ((LOCUS 6)) TASK114317
[17300-C] (REVALUED (COUNT 6 1))
[17800-S] --> GAZE ((FEATURE (SHAPE PLANE))) TASK114316
[18050-C] (NEW (SEARCHMAP 6 ([VOF:{SQ173}])))
[18050-S] --> GAZE ((LOCUS 1A)) TASK114337
[18800-S] --> GAZE ((LOCUS 3)) TASK114347
```





Above: radar display for LAX airspace as it would appear with one plane

Below: same display, annotated with region boundaries, region labels fix names, and airport names.

Finally, the aircraft approaches sufficiently near the LAHAB fix to warrant action. The agent first queries its memory to see whether this routing action has already been done.

```
[72800-S] --> GAZE ((LOCUS 6)) TASK115082
[73550-S] --> GAZE ((FEATURE (SHAPE PLANE))) TASK115081
[73800-C] (IN-RANGE [VOF:{SQ173}] LAHAB)
[73800-S] --> MEMORY ((RETRIEVE (CLEARED [VOF:{SQ173}] ?CALLSIGN DIRECT-TO
    DOWNE))) TASK115101
```

After verifying that the plane has yet to be vectored, a signal is sent to the VOCAL resource to initiate the appropriate clearance. Afterwards, the agent encodes in memory that the clearance has been issued.

```
[76570-S] --> VOCAL ((SQ173 CLEARED DIRECT TO DOWNE)
                     (DIRECT-TO [VOF:{SQ173}] DOWNE)) TASK115156
[76820-R] %vocal% SQ173
[77070-R] %vocal% CLEARED
[77300-S] --> GAZE ((LOCUS 1A)) TASK115194
[77320-R] %vocal% DIRECT
[77570-R] %vocal% TO
[77820-R] %vocal% DOWNE {(DIRECT-TO [VOF:{SQ173}] DOWNE)}
[77820-*] (DIRECT-TO [VOF:{SQ173}] DOWNE)
[77920-S] --> MEMORY ((ENCODE (CLEARED [VOF:{SQ173}] SQ173 DIRECT-TO
    DOWNE))) TASK115155
[78050-S] --> GAZE ((LOCUS 5)) TASK115207
[78520-C] (NEW (CLEARED [VOF:{SQ173}] SQ173 DIRECT-TO DOWNE))
[78550-S] --> MEMORY ((RETRIEVE (EQUIPPED [VOF:{SQ173}] DATALINK
    ?TRUTHVAL))) TASK115222
```

The pilot's compliance with this clearance is indicated on the radar scope approximately 1.5 seconds later. Only a brief time thereafter, the simulated controller detects a change in the orientation of the aircraft's display icon.

```
[80000-W] (HDG 278 306 AC:(SQ173.49-278v26-(45.14 18.36)))
[80050-C] (REVALUED (ORIENTATION [VOF:{SQ173}] 278))
```

Since the aircraft is headed for a landing at LAX, it must be cleared to an altitude of 1900 feet above sea level, the altitude appropriate for a final approach into LAX.

```
[80570-S] --> VOCAL ((SQ173 CHANGE ALTITUDE TO 1900) (ALTITUDE
   [VOF:{SQ173}] 1900)) TASK115248
[80820-R] %vocal% SQ173
[81070-R] %vocal% CHANGE
[81300-S] --> GAZE ((LOCUS 3)) TASK115287
[81320-R] %vocal% ALTITUDE
[81570-R] %vocal% TO
[81820-R] %vocal% 1900 {(ALTITUDE [VOF:{SQ173}] 1900)}
[81820-*] (ALTITUDE [VOF:{SQ173}] 1900)
[81920-S] --> MEMORY ((ENCODE (CLEARED [VOF:{SQ173}] SQ173 ALTITUDE
   1900))) TASK115247
[82520-C] (NEW (CLEARED [VOF:{SQ173}] SQ173 ALTITUDE 1900))
```

Continued scanning reveals, once again, that a plane is close enough to LAHAB for rerouting. This time, a memory retrieval action determines that the routing action has already been performed. No action is taken.

```
[83800-C] (IN-RANGE [VOF:{SQ173}] LAHAB)
[83800-S] --> MEMORY ((RETRIEVE (CLEARED [VOF:{SQ173}] ?CALLSIGN DIRECT-TO
   DOWNE))) TASK115329
```

The plane passes from region 6 into region 5,

```
[164000-W] (LOC (38.21 19.21) (38.54 19.17) AC:(SQ173.1900-277-26-(38.21
   19.21)))
[164050-C] (NEW (INREGION LAX-APPROACH [VOF:{SQ173}] TRUE))
[164050-C] (NEW (INREGION 5 [VOF:{SQ173}] TRUE))
```

eventually coming close enough to DOWNE to begin landing procedures. The agent confirms that these procedures have not already been carried out, selects whether the plane should land on the right or left runway (right), and then vectors the plane to the appropriate initial approach fix (Iaf-Right).

```
[250300-C] (IN-RANGE [VOF:{SQ173}] DOWNE)
[250300-S] --> MEMORY ((RETRIEVE
   (CLEARED [VOF:{SQ173}] ?CALLSIGN DIRECT-TO ?FIX
             (?IF (MEMBER ?FIX '(IAF-L IAF-R)))))) TASK117742
[250900-C] (NOT RETRIEVED
                (CLEARED [VOF:{SQ173}] ?CALLSIGN DIRECT-TO ?FIX
                 (?IF (MEMBER ?FIX '(IAF-L IAF-R)))))
[254300-S] --> VOCAL ((SQ173 CLEARED DIRECT TO IAF-R)
                    (DIRECT-TO [VOF:{SQ173}] IAF-R)) TASK117832
[254550-R] %vocal% SQ173
[254800-R] %vocal% CLEARED
[254950-S] --> GAZE ((FEATURE (SHAPE PLANE))) TASK117851
[255050-R] %vocal% DIRECT
[255300-R] %vocal% TO
[255550-R] %vocal% IAF-R {(DIRECT-TO [VOF:{SQ173}] IAF-R)}
[255550-*] (DIRECT-TO [VOF:{SQ173}] IAF-R)
[255600-S] --> MEMORY ((ENCODE (CLEARED [VOF:{SQ173}] SQ173 DIRECT-TO IAF-
    R))) TASK117831

[256100-C] (REVALUED (ORIENTATION [VOF:{SQ173}] 270))
[256200-C] (REVALUED (CLEARED [VOF:{SQ173}] SQ173 DIRECT-TO IAF-R))
[256200-C] (NEW (CLEARED [VOF:{SQ173}] SQ173 DIRECT-TO IAF-R))
```

Once the aircraft approaches near enough to Iaf-R, it is vectored toward LAX

```
[375550-S] --> VOCAL ((SQ173 CLEARED DIRECT TO LAX) (DIRECT-TO
    [VOF:{SQ173}] LAX)) TASK119681
[375550-S] --> GAZE ((LOCUS 4)) TASK119701
[375800-R] %vocal% SQ173
[376050-R] %vocal% CLEARED
[376300-R] %vocal% DIRECT
[376550-R] %vocal% TO
[376800-R] %vocal% LAX {(DIRECT-TO [VOF:{SQ173}] LAX)}
[376800-*] (DIRECT-TO [VOF:{SQ173}] LAX)
[376850-S] --> MEMORY ((ENCODE (CLEARED [VOF:{SQ173}] SQ173 DIRECT-TO
    LAX))) TASK119680
[380000-W] (HDG 241 270 AC:(SQ173.1900-241v26-(20.39 20.13)))
[380050-C] (REVALUED (ORIENTATION [VOF:{SQ173}] 241))
```

and eventually told to contact the LAX tower controller for permission to land (i.e. it is handed off to tower).

```
[507800-C] (IN-RANGE [VOF:{SQ173}] LAX)
[507800-S] --> MEMORY ((RETRIEVE (CLEARED [VOF:{SQ173}] ?CALLSIGN HANDOFF
    LAX))) TASK121678
[508400-C] (NOT RETRIEVED (CLEARED [VOF:{SQ173}] ?CALLSIGN HANDOFF LAX))

[510550-S] --> VOCAL ((SQ173 CONTACT LAX GOOD DAY) (HANDOFF [VOF:{SQ173}]
    LAX)) TASK121731
[510800-R] %vocal% SQ173
[511050-R] %vocal% CONTACT
[511300-R] %vocal% LAX
[511550-R] %vocal% GOOD
[511800-R] %vocal% DAY {(HANDOFF [VOF:{SQ173}] LAX)}
[511800-*] (HANDOFF [VOF:{SQ173}] LAX)
[511850-S] --> MEMORY ((ENCODE (CLEARED [VOF:{SQ173}] SQ173 HANDOFF LAX)))
    TASK121730
```

As control passes to the tower controller, the aircraft becomes gray on the radar display, and eventually disappears entirely.

```
[512000-W] (WHO-CONTROLS LAX TRACON AC:(SQ173t1900-240-26-(10.82 14.53)))
[512100-C] (REVALUED (COLOR [VOF:{SQ173}] GRAY))
[512450-C] (NEW (CLEARED [VOF:{SQ173}] SQ173 HANDOFF LAX))
[540050-C] (EXTINGUISHED (INREGION 4BL [VOF:{SQ173}] TRUE))
[540050-C] (EXTINGUISHED (INREGION 4B [VOF:{SQ173}] TRUE))
[568050-C] (EXTINGUISHED (INREGION 4A [VOF:{SQ173}] TRUE))
[568050-C] (EXTINGUISHED (INREGION 4 [VOF:{SQ173}] TRUE))
[568050-C] (EXTINGUISHED (INREGION LAX-APPROACH [VOF:{SQ173}] TRUE))
```

A.3.2 Examining a high-detail, short timeline trace

```
----- Starting simulation run -----

1214 ARRIVAL AC:(DL603c54-180-23-(43.700787 37.598423))
```

At the beginning of a simulation run, the root task *{do-domain}* enables a set of tasks, most of which prepare the agent to respond when certain events occur.

```
[200-A] EXECUTING [TASK5598 (DO-DOMAIN) {ENABLED}]
[200-A] ..CREATING [TASK5601 (MONITOR-ATC-EXPECTATIONS) {NIL}]
[200-A] ..CREATING [TASK5602 (HANDLE-WAYPOINTS) {NIL}]
[200-A] ..CREATING [TASK5603 (SCAN-RADAR-DISPLAY) {NIL}]
[200-A] ..CREATING [TASK5604 (HANDLE-NEW-PLANES) {NIL}]
```

An initial ATC task called *{handle-new-planes}* generates and enables a subtask called *{handle-new-plane}* using the following procedure:

```
(procedure
 (index (handle-new-planes))
 (step s1 (handle-new-plane ?plane)
   (period :recurrent :reftime enabled)
   (waitfor (new (shape ?plane ?shapes (?if (member ‘plane ?shapes)))))))
```

This prepares the simulated controller to respond whenever visual processing generates an event of the form *(new (shape ?plane ?shapes))* where *?shapes* is a list including the symbol *plane*. When the task is created, the precondition of having detected such an event has not been met, so the task is left in a pending state.

```
[200-A] TESTING preconditions for
    [TASK5604 (HANDLE-NEW-PLANES) {PENDING}]....  SATISFIED
[200-A] SELECTING procedure for TASK5604...  => (HANDLE-NEW-PLANES)
[200-A] ENABLING [TASK5604 (HANDLE-NEW-PLANES) {ENABLED}]
[200-A] EXECUTING [TASK5604 (HANDLE-NEW-PLANES) {ENABLED}]
[200-A] ..CREATING [TASK5623 (HANDLE-NEW-PLANE ?PLANE) {NIL}]
[200-A] TESTING preconditions for
    [TASK5623 (HANDLE-NEW-PLANE ?PLANE) {PENDING}]....  NOT-SATISFIED
```

When a new plane arrives and is detected, in this case an aircraft with the callsign DL603, an arrival altitude of 5400 feet above sea level, a heading of 180 degrees, and an airspeed of 230 knots,

```
[4000-W] (ARRIVAL AC:(DL603c54-180-23-(43.700787 37.598423)))
[4100-C] (NEW (SHAPE [VOF:{DL603}] (PLANE ICON VISOB)))
[4100-C] (NEW (BLINK [VOF:{DL603}] 2))
[4100-C] (NEW (ORIENTATION [VOF:{DL603}] 180))
[4100-C] (NEW (COLOR [VOF:{DL603}] GREEN))
```

the agent responds by enabling the plane-handling task. Since the task is recurrent with its reference for reinstantiation set to enable-time, enablement causes a copy of the task to be created, thus preparing a response to any subsequent arrival.

```
[4220-A] TESTING preconditions for
    [TASK5623 (HANDLE-NEW-PLANE [VOF:{DL603}]) {PENDING}]....  SATISFIED
[4220-A] SELECTING procedure for TASK5623...  => (HANDLE-NEW-PLANE ?PLANE)
[4220-A] ENABLING [TASK5623 (HANDLE-NEW-PLANE [VOF:{DL603}]) {ENABLED}]

[4220-A] ..CREATING [TASK5713 (HANDLE-NEW-PLANE [VOF:{DL603}]) {NIL}]
[4220-A] TESTING preconditions for
    [TASK5713 (HANDLE-NEW-PLANE [VOF:{DL603}]) {PENDING}]....  NOT-
   SATISFIED
```

The task is then executed by instantiating subtasks as prescribed by the selected procedure:

```
(procedure
    (index (handle-new-plane ?plane))
    (profile (gaze 8 10))
    (interrupt-cost 6)
    (step s1 (doubleclick-icon ?plane))
    (step s2 (vis-examine ?plane 500))
    (step s3 (determine callsign for ?plane => ?callsign) (waitfor ?s2))
    (step s4 (encode (callsign ?plane ?callsign)) (waitfor ?s3))
    (step s5 (determine destination ?plane => ?destination) (waitfor ?s4))
    (step s6 (encode (destination ?plane ?destination)) (waitfor ?s5))
    (step s7 (terminate) (waitfor ?s6)))
```

```
[4220-A] EXECUTING [TASK5623 (HANDLE-NEW-PLANE [VOF:{DL603}]) {ENABLED}]
[4220-A] ..CREATING [TASK5714 (TERMINATE ?SELF SUCCESS >> NIL) {NIL}]
[4220-A] ..CREATING [TASK5715 (ENCODE (DESTINATION [VOF:{DL603}]
    ?DESTINATION)) {NIL}]
[4220-A] ..CREATING [TASK5716 (DETERMINE DESTINATION [VOF:{DL603}]) {NIL}]
[4220-A] ..CREATING [TASK5717 (ENCODE (CALLSIGN [VOF:{DL603}] ?CALLSIGN))
    {NIL}]
[4220-A] ..CREATING [TASK5718 (DETERMINE CALLSIGN FOR [VOF:{DL603}])
    {NIL}]
[4220-A] ..CREATING [TASK5719 (VIS-EXAMINE [VOF:{DL603}] 500) {NIL}]
[4220-A] ..CREATING [TASK5720 (DOUBLECLICK-ICON [VOF:{DL603}]) {NIL}]
```

Executing the procedure generates 7 new tasks. Two of these, one to visually examine the aircraft icon and one to double-click on it have their non-resource preconditions satisfied at the outset. The others are defined to wait until one of their sibling tasks terminates. These fail their initial precondition test. Note that tests for precondition satisfaction, especially tests that fail, are quite common. For clarity, most testing events have been edited out of the trace below.

```
[4220-A] TESTING preconditions for
    [TASK5720 (DOUBLECLICK-ICON [VOF:{DL603}]) {PENDING}]....  SATISFIED
[4220-A] SELECTING procedure for TASK5720...  => (DOUBLECLICK-ICON ?ICON)
[4220-A] ENABLING [TASK5720 (DOUBLECLICK-ICON [VOF:{DL603}]) {ENABLED}]
[4220-A] TESTING preconditions for
    [TASK5719 (VIS-EXAMINE [VOF:{DL603}] 500) {PENDING}]....  SATISFIED
[4220-A] SELECTING procedure for TASK5719...  => (VIS-EXAMINE ?ITEM ?TIME)
[4220-A] TESTING preconditions for
```

```
      [TASK5718 (DETERMINE CALLSIGN FOR [VOF:{DL603}]) {PENDING}]....  NOT-
    SATISFIED
[4220-A] TESTING preconditions for
      [TASK5717 (ENCODE (CALLSIGN [VOF:{DL603}] ?CALLSIGN)) {PENDING}]....
    NOT-SATISFIED
[4220-A] TESTING preconditions for
      [TASK5716 (DETERMINE DESTINATION [VOF:{DL603}]) {PENDING}]....  NOT-
    SATISFIED
[4220-A] TESTING preconditions for
      [TASK5715 (ENCODE (DESTINATION [VOF:{DL603}] ?DESTINATIO
    {PENDING}].... NOT-SATISFIED
[4220-A] TESTING preconditions for
      [TASK5714 (TERMINATE TASK5623 SUCCESS >> NIL) {PENDING}]....  NOT-
    SATISFIED
```

The remainder of the trace will focus on carrying out the task of double-clicking on the aircraft icon. The task begins by instantiating double-click subtasks according to the selected procedure:

```
(procedure
 (index (doubleclick-icon ?icon))
 (step s1 (select-hand-for-mouse => ?hand))
 (step s2 (move pointer to ?icon using ?hand) (waitfor ?s1))
 (step s3 (doubleclick-mouse ?hand) (waitfor ?s2))
 (step s4 (terminate) (waitfor ?s3)))
```

```
[4220-A] EXECUTING [TASK5720 (DOUBLECLICK-ICON [VOF:{DL603}]) {ENABLED}]
[4220-A] ..CREATING [TASK5721 (TERMINATE ?SELF SUCCESS >> NIL) {NIL}]
[4220-A] ..CREATING [TASK5722 (DOUBLECLICK-MOUSE ?HAND) {NIL}]
[4220-A] ..CREATING [TASK5723 (MOVE POINTER TO [VOF:{DL603}] USING ?HAND)
    {NIL}]
[4220-A] ..CREATING [TASK5724 (SELECT-HAND-FOR-MOUSE) {NIL}]
```

The first step in the double-click procedure is to select a hand to perform the action. The following simple procedure always selects the left hand, returning the value *left* upon termination.

```
(procedure
 (index (select-hand-for-mouse))
 (step s1 (terminate >> left)))
```

```
[4220-A] TESTING preconditions for
    [TASK5724 (SELECT-HAND-FOR-MOUSE) {PENDING}]....  SATISFIED
[4220-A] SELECTING procedure for TASK5724...  => (SELECT-HAND-FOR-MOUSE)
[4220-A] ENABLING [TASK5724 (SELECT-HAND-FOR-MOUSE) {ENABLED}]

[4220-A] EXECUTING [TASK5724 (SELECT-HAND-FOR-MOUSE) {ENABLED}]
[4220-A] ..CREATING [TASK5725 (TERMINATE ?SELF SUCCESS >> LEFT) {NIL}]
[4220-A] TESTING preconditions for
    [TASK5725 (TERMINATE TASK5724 SUCCESS >> LEFT) {PENDING}]....
    SATISFIED
[4220-A] ENABLING [TASK5725 (TERMINATE TASK5724 SUCCESS >> LEFT)
    {ENABLED}]
[4220-A] EXECUTING [TASK5725 (TERMINATE TASK5724 SUCCESS >> LEFT)
    {ENABLED}]
[4220-C] (TERMINATE [TASK5724 (SELECT-HAND-FOR-MOUSE) {ONGOING}] SUCCESS)
[4220-C] (TERMINATE [TASK5725 (TERMINATE TASK5724 SUCCESS >> LEFT)
    {ENABLED}] SUCCESS)
```

After selecting a hand, the next step is to use the mouse to move the pointer onto the target icon. This is accomplished by executing the procedure below. The non-resource preconditions for executing the procedure are met as soon as a hand is selected..

```
(procedure
 (index (move pointer to ?icon using ?hand))
 (profile (?hand 8 10) (gaze 8 10))
 (step s1 (grasp ?hand mouse))
 (step s2 (find mouse pointer => ?pointer))
 (step s3 (shift-gaze-to ?icon) (waitfor ?s2))
 (step s4 (compute-pointer-icon-distance ?icon ?pointer => ?dist) (waitfor ?s3))
 (step s5 (map eye-target ?icon to hand-target => ?target) (waitfor ?s2 ?s4))
 (step s6 (move-mouse ?hand ?target) (waitfor ?s5 ?s1))
 (step s7 (terminate)
    (waitfor (completed ?s6))
    (waitfor ?s4 :and (equal ?dist 0.0)))
 (step s8 (reset ?self) (waitfor (resumed ?self))))
```

```
[4220-A] TESTING preconditions for
      [TASK5723 (MOVE POINTER TO [VOF:{DL603}] USING LEFT) {PENDING}]....
    SATISFIED
[4220-A] SELECTING procedure for TASK5723...  => (MOVE POINTER TO ?ICON
    USING ?HAND)
```

but the task is delayed while waiting to gain control of a needed resource from some higher priority task. It then executes, creating a number of subtasks.

```
[5050-A] ENABLING [TASK5723 (MOVE POINTER TO [VOF:{DL603}] USING LEFT)
    {ENABLED}]
[5050-A] EXECUTING [TASK5723 (MOVE POINTER TO [VOF:{DL603}] USING LEFT)
    {ENABLED}]
[5050-A] ..CREATING [TASK5732 (RESET ?SELF) {NIL}]
[5050-A] ..CREATING [TASK5733 (TERMINATE ?SELF SUCCESS >> NIL) {NIL}]
[5050-A] ..CREATING [TASK5734 (MOVE-MOUSE LEFT ?TARGET) {NIL}]
[5050-A] ..CREATING [TASK5735 (MAP EYE-TARGET [VOF:{DL603}] TO HAND-
    TARGET) {NIL}]
[5050-A] ..CREATING [TASK5736 (COMPUTE-POINTER-ICON-DISTANCE [VOF:{DL603}]
    ?POINTER) {NIL}]
[5050-A] ..CREATING [TASK5737 (SHIFT-GAZE-TO [VOF:{DL603}]) {NIL}]
[5050-A] ..CREATING [TASK5738 (FIND MOUSE POINTER) {NIL}]
[5050-A] ..CREATING [TASK5739 (GRASP LEFT MOUSE) {NIL}]
```

The task of grasping the mouse is enabled, the main result of which is a signal to the LEFT hand resource to carry a grasp action.

```
[5050-A] TESTING preconditions for
    [TASK5739 (GRASP LEFT MOUSE) {PENDING}]....  SATISFIED
[5050-A] SELECTING procedure for TASK5739...  => (GRASP ?HAND ?OBJECT)
[5050-A] ENABLING [TASK5739 (GRASP LEFT MOUSE) {ENABLED}]
[5050-A] EXECUTING [TASK5739 (GRASP LEFT MOUSE) {ENABLED}]
[5050-A] ..CREATING [TASK5745 (TERMINATE ?SELF SUCCESS >> NIL) {NIL}]
[5050-A] ..CREATING [TASK5746 (RESET ?SELF) {NIL}]
[5050-A] ..CREATING [TASK5747 (SIGNAL-RESOURCE LEFT (GRASP MOUSE)) {NIL}]
[5050-A] ..CREATING [TASK5748 (CLEAR-HAND LEFT) {NIL}]
[5050-A] ..CREATING [TASK5749 (GRASP-STATUS LEFT MOUSE) {NIL}]

[5050-A] ENABLING [TASK5747 (SIGNAL-RESOURCE LEFT (GRASP MOUSE))
    {ENABLED}]
[5050-A] EXECUTING [TASK5747 (SIGNAL-RESOURCE LEFT (GRASP MOUSE))
    {ENABLED}]
[5050-S] --> LEFT ((GRASP MOUSE)) TASK5739
[5050-C] (TERMINATE [TASK5747 (SIGNAL-RESOURCE LEFT (GRASP MOUSE))
    {ENABLED}] SUCCESS)
```

Concurrently, a task to shift gaze to the double-click target is initiated

```
[5050-A] ENABLING [TASK5737 (SHIFT-GAZE-TO [VOF:{DL603}]) {ENABLED}]
[5050-A] EXECUTING [TASK5737 (SHIFT-GAZE-TO [VOF:{DL603}]) {ENABLED}]
```

which, via a series of intermediate procedures, produces a signal to the GAZE resource.

```
[5050-A] EXECUTING [TASK5744 (SIGNAL-RESOURCE GAZE (LOCUS [VOF:{DL603}]))
    {ENABLED}]
[5050-S] --> GAZE ((LOCUS [VOF:{DL603}])) TASK5741
[5050-C] (TERMINATE [TASK5744 (SIGNAL-RESOURCE GAZE (LOCUS [VOF:{DL603}]))
    {ENABLED}] SUCCESS)
```

The agent prepares to move the mouse by verifying that the location of the mouse pointer is known by

```
[5050-A] EXECUTING [TASK5738 (FIND MOUSE POINTER) {ENABLED}]
[5050-C] (TERMINATE [TASK5738 (FIND MOUSE POINTER) {ENABLED}] SUCCESS)
```

waiting for the gaze shift to the target to complete (thus providing specific location information),

```
[5200-C] (COMPLETED [TASK5741 (VIS-EXAMINE [VOF:{DL603}] 0) {ONGOING}])
```

computing a hand motion that will bring the pointer to the target,

```
[5450-A] EXECUTING [TASK5735 (MAP EYE-TARGET [VOF:{DL603}] TO HAND-TARGET)
    {ENABLED}]
[5450-C] (TERMINATE [TASK5735 (MAP EYE-TARGET [VOF:{DL603}] TO HAND-
    TARGET) {ENABLED}] SUCCESS)
```

and waiting for the mouse to be in hand.

```
[6550-*] (GRASP MOUSE)
[6550-C] (COMPLETED [TASK5739 (GRASP LEFT MOUSE) {ONGOING}])
```

Then the mouse move action can proceed, resulting in an appropriate signal to the LEFT hand

```
[6650-A] TESTING preconditions for
    [TASK5734 (MOVE-MOUSE LEFT {DL603}) {PENDING}]....  SATISFIED
[6650-A] SELECTING procedure for TASK5734...  => (MOVE-MOUSE ?HAND
    ?TARGET)
[6650-A] ENABLING [TASK5734 (MOVE-MOUSE LEFT {DL603}) {ENABLED}]
```

```
[6650-A] EXECUTING [TASK5734 (MOVE-MOUSE LEFT {DL603}) {ENABLED}]
[6650-A] ..CREATING [TASK5750 (RESET ?SELF) {NIL}]
[6650-A] ..CREATING [TASK5751 (TERMINATE ?SELF SUCCESS >> NIL) {NIL}]
[6650-A] ..CREATING [TASK5752 (SIGNAL-RESOURCE LEFT (MOVE POINTER
    {DL603})) {NIL}]

[6650-A] ENABLING [TASK5752 (SIGNAL-RESOURCE LEFT (MOVE POINTER {DL603}))
    {ENABLED}]
[6650-A] EXECUTING [TASK5752 (SIGNAL-RESOURCE LEFT (MOVE POINTER {DL603}))
    {ENABLED}]
[6650-S] --> LEFT ((MOVE POINTER {DL603})) TASK5734
[6650-C] (TERMINATE [TASK5752 (SIGNAL-RESOURCE LEFT (MOVE POINTER
    {DL603})) {ENABLED}] SUCCESS)
```

which eventually results in a successfully completed motion.

```
[8150-*] (MOVE POINTER {DL603})
[8150-C] (COMPLETED [TASK5734 (MOVE-MOUSE LEFT {DL603}) {ONGOING}])
```

Finally, the double-click task can be carried out.

```
[8250-A] TESTING preconditions for
    [TASK5722 (DOUBLECLICK-MOUSE LEFT) {PENDING}]....  SATISFIED
[8250-A] SELECTING procedure for TASK5722...  => (DOUBLECLICK-MOUSE ?HAND)
[8250-A] ENABLING [TASK5722 (DOUBLECLICK-MOUSE LEFT) {ENABLED}]
[8250-A] EXECUTING [TASK5722 (DOUBLECLICK-MOUSE LEFT) {ENABLED}]
[8250-A] ..CREATING [TASK5761 (RESET ?SELF) {NIL}]
[8250-A] ..CREATING [TASK5762 (TERMINATE ?SELF SUCCESS >> NIL) {NIL}]
[8250-A] ..CREATING [TASK5763 (TERMINATE ?SELF FAILURE >> NIL) {NIL}]
[8250-A] ..CREATING [TASK5764 (SIGNAL-RESOURCE LEFT (DOUBLECLICK)) {NIL}]
[8250-A] ..CREATING [TASK5765 (VERIFY-RESOURCE-STATE LEFT GRASP MOUSE)
    {NIL}]
```

This involves first verifying that the hand which is to perform the double-click action is currently grasping a mouse (yes),

```
[8250-A] ENABLING [TASK5765 (VERIFY-RESOURCE-STATE LEFT GRASP MOUSE)
    {ENABLED}]
[8250-A] EXECUTING [TASK5765 (VERIFY-RESOURCE-STATE LEFT GRASP MOUSE)
    {ENABLED}]
[8250-C] (TERMINATE [TASK5765 (VERIFY-RESOURCE-STATE LEFT GRASP MOUSE)
    {ENABLED}] SUCCESS)
```

and then signaling the LEFT hand resource to do the action.

```
[8250-A] ENABLING [TASK5764 (SIGNAL-RESOURCE LEFT (DOUBLECLICK))
    {ENABLED}]
[8250-A] EXECUTING [TASK5764 (SIGNAL-RESOURCE LEFT (DOUBLECLICK))
    {ENABLED}]
[8250-S] --> LEFT ((DOUBLECLICK)) TASK5722
[8250-C] (TERMINATE [TASK5764 (SIGNAL-RESOURCE LEFT (DOUBLECLICK))
    {ENABLED}] SUCCESS)
```

The action succeeds a brief time afterwards

```
[9750-*] (DOUBLECLICK)
[9750-C] (COMPLETED [TASK5722 (DOUBLECLICK-MOUSE LEFT) {ONGOING}])
```

thereby causing the task and its parent task to terminate. Note that the termination of one task often causes the termination of others (siblings and parents), leading to termination cascades in the simulation trace.

```
[9850-A] TESTING preconditions for
    [TASK5762 (TERMINATE TASK5722 SUCCESS >> NIL) {PENDING}]....
    SATISFIED
[9850-A] ENABLING [TASK5762 (TERMINATE TASK5722 SUCCESS >> NIL) {ENABLED}]
[9850-A] EXECUTING [TASK5762 (TERMINATE TASK5722 SUCCESS >> NIL)
    {ENABLED}]
[9850-C] (TERMINATE [TASK5722 (DOUBLECLICK-MOUSE LEFT) {ONGOING}] SUCCESS)
[9850-C] (TERMINATE [TASK5761 (RESET TASK5722) {PENDING}] SUCCESS)
[9850-C] (TERMINATE [TASK5762 (TERMINATE TASK5722 SUCCESS >> NIL)
    {ENABLED}] SUCCESS)
[9850-A] TESTING preconditions for
    [TASK5721 (TERMINATE TASK5720 SUCCESS >> NIL) {PENDING}]....
    SATISFIED
[9850-A] ENABLING [TASK5721 (TERMINATE TASK5720 SUCCESS >> NIL) {ENABLED}]
[9850-A] EXECUTING [TASK5721 (TERMINATE TASK5720 SUCCESS >> NIL)
    {ENABLED}]
[9850-C] (TERMINATE [TASK5720 (DOUBLECLICK-ICON [VOF:{DL603}]) {ONGOING}]
    SUCCESS)
[9850-C] (TERMINATE [TASK5721 (TERMINATE TASK5720 SUCCESS >> NIL)
    {ENABLED}] SUCCESS)
```

After completing the double-click, several more plane-handling actions are carried out according to the selected plane-handling procedure. When these are complete, the plane-handling task terminates.

```
[11050-A] ENABLING [TASK5714 (TERMINATE TASK5623 SUCCESS >> NIL)
    {ENABLED}]
```

```
[11050-A] EXECUTING [TASK5714 (TERMINATE TASK5623 SUCCESS >> NIL)
    {ENABLED}]
[11050-C] (TERMINATE [TASK5623 (HANDLE-NEW-PLANE [VOF:{DL603}]) {ONGOING}]
    SUCCESS)
[11050-C] (TERMINATE [TASK5714 (TERMINATE TASK5623 SUCCESS >> NIL)
    {ENABLED}] SUCCESS)
```

References

Agre, P.E. and Chapman, D. (1987) Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 268-272.

Anderson, J.R. (1991) The Place of Cognitive Architectures in a Rational Analysis. In *Architectures for Intelligence / the Twenty-second Carnegie Symposium on Cognition.* Edited by Kurt VanLehn. Hillsdale, NJ: Lawrence Earlbaum Associates

Anderson, J.R. (1990). *The adaptive character of thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, J.R. (1982) Acquisition of Cognitive Skill. *Psychological Review*, 89, 369-406.

Baddeley, A. (1990) *Human Memory / Theory and Practice*. Needham Heights, MA: Simon and Schuster.

Baecker, R.M., Grudin, J., Buxton, W.A.S., and Greenberg, S. (1995) *Human-Computer Interaction: Toward the Year 2000.* San Francisco, CA: Morgan Kaufmann.

Byrne, M.D. and Bovair, S. (1997) A working memory model of a common procedural error, *Cognitive Science*, 22(1).

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

Carrier, L.M. and Pashler, H. (1995) Attentional limitations in memory retrieval. *Journal for experimental psychology: learning, memory, & cognition*, 21, 1339-1348.

Chappell, S.L. (1994) Using Voluntary Incident Reports for Human Factors Evaluations. In N. Johnston, N. McDonald and R. Fuller (Eds.), *Aviation Psychology in Practice*. Aldershot, England: Ashgate.

Cohen, P.R., Greenberg, M.L., Hart, D. and Howe, A.E. (1989) An Introduction to Phoenix, the EKSL Fire-Fighting System. EKSL Technical Report. Department of Computer and Informational Science. University of Massachusetts, Amherst.

Corker, K.M. and Smith, B.R. (1993) An architecture and model for cognitive engineering simulation analysis. *Proceedings of the AIAA Computing in Aerospace 9 Conference*, San Diego, CA.

Curtis, B., Krasner, H. and Iscoe, N. (1981) A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, 31 (11), 1268-1287.

Degani, A. (1996). *Modeling human-machine systems: On modes, error, and patterns of interaction*. Ph.D. thesis. Atlanta, GA: Georgia Institute of Technology.

Degani, A., & Shafto, M.G. (1997). *Process algebra and human-machine modeling*. Cognitive Technology Conference. Terre Haute, IN: Indiana State University.

DeGroot, A.D. (1978) *Thought and choice in chess* (2nd ed.) The Hague: Mouton.

Deutsch, S.E., Adams, M.J., Abrett, G.A., Cramer, N.L., and Freeher, C.E. (1993) RDT&E Support: Operator Model Architecture (OMAR) Software Functional Specification (AL/HR-TP-1993-0027), Wright-Patterson AFB, OH: Armstrong Laboratory, Logistics Research Division.

Ellis, S. and Stark, L. (1986) Statistical Dependency in Visual Scanning. *Human Factors*, 28 (4), 421-438.

Ericsson, K.A., & Smith, J.A. (Eds.). (1991). *Toward a general theory of expertise*. Cambridge, UK: Cambridge University Press.

Firby, R.J. (1989) *Adaptive execution in complex dynamic worlds*. Ph.D. thesis, Yale University.

Fitts, P.M. and Peterson, J.R. (1964) Information capacity of discrete motor responses. *Journal of Experimental Psychology*, 67, 103-112.

Freed, M.A. (1996). Using the RAP system to predict human error. In *Proceedings of the 1996 AAAI Symposium on Plan Execution: Problems and Issues*. AAAI Press.

Freed, M.A. & Remington, R.W. (1997). Managing decision resources in plan execution. In *Proceedings of the Fifteenth Joint Conference on Artificial Intelligence*. Nagoya, Japan.

Freed, M. and Shafto, M. (1997) Human-system modelling: some principles and a pragmatic approach. *Proceedings of the Fourth International Workshop on the Design, Specification, and Verification of Interactive System*. Granada, Spain.

Friedland, P.E. (1979) *Knowledge-based Experiment Design in Molecular Genetics*. Report number 79-711 (doctoral dissertation), Computer Science Department, Stanford University.

Gat, Erann. (1992) Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots. In *Proceedings of the 1992 National Conference on Artificial Intelligence*.

Georgeff, M. and Lansky,A. (1987) Reactive Reasoning and Planning: An Experiment with a Mobile Robot. *Proceedings of the 1987 National Conference on Artificial Intelligence*.

Gick and Holyoak (1987) The Cognitive Basis of Knowledge Transfer. In S.M. Cormier & J.D. Hagman (Eds.), *Transfer of Learning: Contemporary Reasearch and Applications*, pp. 9-46, San Diego, CA: Academic Press.

Gould, J.D. (1988). How to design usable systems. In M. Helander (Ed.), *Handbook of Human-Computer Interaction*. New York: North-Holland.

Gray, W.D., John, B.E., & Atwood, M.E. (1993). Project Ernestine: Validating a GOMS analysis for predicting and explaining real-world task performance. *Human Computer Interaction*, 8, 237-309.

Hart, S. and Bortolussi, M.R. (1983) Pilot Errors as a Source of Workload. Paper presented at the Second Symposium on Aviation Psychology, Columbus, OH.

Halverson, C.A. (1995). *Inside the cognitive workplace: New technology and air traffic control*. Unpublished doctoral dissertation. La Jolla, CA: University of California, San Diego.

Hayes-Roth, B. (1995) An Architecture for Adaptive Intelligent Systems. *Artificial Intelligence*, 72, 329-365.

Hutchins, E. (1995) How a cockpit remembers its speed. *Cognitive Science*, 19(3), 265-288.

John, B. and Vera, A. (1992) A GOMS analysis of a graphic, machine-paced, highly interactive task. *Proceedings CHI'92*, ACM, 251-258.

John, B.E. and Kieras, D.E. (1994) The GOMS Family of Analysis Techniques: Tools for Design and Evaluation. Carnegie Mellon University. School of Computer Science, TR CMU-CS-94-181.

John, B.E. and Newell, A. (1989) Cumulating the Science of HCI: From S-R Compatibility to Transcription Typing. *Proceedings of CHI'89 Conference on Human Factors in Computing Systems*, 109-114, New York: ACM.

Just, M.A. and Carpenter, P.A. (1992) A capacity theory of comprehension: Individual differences in working memory. *Psychological Review*, 99, 123-148.

Kahneman, D., Triesman, A. and Gibbs, B.J. (1992) The Reviewing of Object Files: Object-Specific Integration of Information. *Cognitive Psychology*, 24, 175-219.

Kieras, D.E. and Meyer, D.E. (1994) The EPIC architecture for modeling human information-processing: A brief introduction. (EPIC Tech Re. No. 1, TR-94/ONR-EPIC-1). Ann Arbor, University of Michigan, Department of Electrical Engineering and Computer Science.

Kitsch, W. (1988) The role knowledge in discourse comprehension: A construction-integration model. *Psychological Review*, 95, 163-182.

Kirwan, B. and Ainsworth, L. (1992) *A Guide to Task Analysis*. Taylor and Francis.

Kitajima and Polson (1995) A comprehension-based model of correct performance and errors in skilled, display-based human-computer interaction. *International Journal of Human-Computer Systems*, 43, 65-69.

Laird, J.E., Newell, A., Rosenbloom, P.S. (1987) SOAR: An Architecture for General Intelligence. *Artificial Intelligence*, 33, 1-64.

T.K. Landauer. (1991) Let's Get Real: A Position Paper on the Role of Cognitive Psychology in the Design of Humanly Useful and Usable Systems. In J.M. Carroll (Ed.), *Designing Interaction*, Cambridge University Press, pp. 60-73.

Leveson, N. (1995) *Safeware: System Safety and Computers*. Addison Wesley Publishing Co.

Lewis, R.L., & Polk, T.A. (1994). *Preparing to Soar: Modeling pilot cognition in the Taxi-MIDAS Scenario.* Cognitive Modeling Workshop. Moffett Field, CA: NASA-Ames Research Center.

MacMillan, J., Deutch, S.E, and Young, M.J. (1997) A comparison of alternatives for automated decision support in a multi-task environment. *Proceedings of the Human Factors and Ergonomics Society 41st Annual Meeting, Albuquerque*, NM, September 22-26.

May, J., Blandford, A., Barnard, P., & Young, R. (1994). *Interim report on the application of user modelling techniques to the shared exemplars*. AMODEUS Project Document D6. Cambridge, UK: MRC Applied Psychology Unit.

McCarthy, J. and Hayes, P. (1969) Some Philosophical Problems from the Standpoint of Artificial Intelligence. In Meltzer, B. and Richie, D. (eds.), *Machine Intelligence* 4, Edinburgh, UK: Edinburgh University Press.

McKee, S.D. (1991) A Local Mechanism for Differential Velocity Detection. *Vision Research*, 21, 491-500.

Mentemerlo, M.D. and Eddowes, E. (1978) The judgemental nature of task analysis. In *Proceedings of the Human Factors Society*, pp. 247-250, Santa Monica, CA.

Mills, T.S. and Archibald, J.S. (1992) *The Pilot's Reference to ATC Procedures and Terminology*. Reavco Publishing, Van Nuys, CA.

Mosier, J. and Smith, S. (1986) Applications of Guidelines for Designing User Interface Software. *Behavior and Information Technology* 5(1), 39-46.

Newell, A. (1990) *Unified theories of cognition*. Cambridge, Mass; Harvard University Press.

Nielsen, J. (1993) *Usability Engineering*. Academic Press.

Norman, D. A. (1998) *The Psychology of Everyday Things*. New York, N.Y: Basic Books.

Norman, D.A. (1981) Categorization of Action Slips. *Psychological Review*, 88, 1-15.

NTSB (1986) Runway Incursions at Controlled Airports in the United States. NTSB/SIR-86/01.

Olson, J.R. and Olson G.M. (1989) The growth of cognitive modeling in human-computer interaction since GOMS. *Human Computer Interaction*.

Owens, C. (1990) Representing Abstract Plan Failures. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, Lawrence Earlbaum Associates, 277-284.

Payne, John W., Bettman, James, R., and Johnson, Eric J. (1993) *The adaptive decision maker*. Cambridge University Press.

Pell, B., Bernard, D.E., Chien, S.A.., Gat, E., Muscettola, N., Nayak, P.P., Wagner, M., and Williams, B.C. (1997) An autonomous agent spacecraft prototype. *Proc. of the First International Conference on Autonomous Agents*, ACM Press.

Polson, P., Lewis, C., Rieman, J., Wharton, C., and Wilde, N. (1992) Cognitive Walkthroughs: A method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36, 741-773.

Reason, J.T. (1990) *Human Error*. Cambridge University Press, New York, N.Y.

Reason, J.T. and Mycielska, K. (1982) *Absent-minded? The psychology of mental lapses and everyday errors*. Englewood Cliffs, N.J., Prentice Hall.

Remington, R.W., Johnston, J.C., and Yantis, S. (1992) Involuntary Attentional Capture by Abrupt Onsets. *Perception and Psychophysics*, 51 (3), 279-290.

Remington, Roger W., Johnston, James C., Bunzo, Marilyn S., and Benjamin, Kirk A. (1990) The Cognition Simulation System: An Interactive Graphical Tool for Modeling Human Cognitive Processing. In Object-Oriented Simulation, San Diego: Society for Computer Simulation, pp. 155-166.

Remington, R.W., & Shafto, M.G. (1990). *Building human interfaces to fault diagnostic expert systems I: Designing the human interface to support cooperative fault diagnosis*. Seattle, WA: CHI'90 Workshop on Computer-Human Interaction in Aerospace Systems.

Rogers, W.A. and Fisk, A.D. (1997) ATM Design and Training Issues. *Ergonomics and Design*, 5 (1), 4-9.

Rosenbloom, P.S., Newell, A., and Laird, J.E. (1991) Toward the Knowledge Level in Soar: The Role of the Architecture in the Use of Knowledge. In *Architectures for Intelligence / the Twenty-second Carnegie Symposium on Cognition*. Edited by VanLehn, Kurt, Hillsdale, NJ: Lawrence Earlbaum Associates.

Salthouse, T.A. (1991) Expertise as the circumvention of human processing limitations. In Ericsson, K.A., & J.A. Smith (Eds.), *Toward a general theory of expertise*. Cambridge, UK: Cambridge University Press.

Schank, Roger C. (1986) *Explanation Patterns*. Hillsdale, NJ: Lawrence Earlbaum Associates:

Schneider, W. and Detweiler, M. (1988) The role of practice in dual-task performance: Toward Workload Modeling in a Connectionist/Control Architecture. *Human Factors*, 30(5), 539-566.

Schneiderman, B. (1992) Designing the User Interface: Strategies for Effective Human-Computer Interaction, Second edition, Addison-Wesley.

Seifert, C.M., & Shafto, M.G. (1994). Computational models of cognition. In J. Hendler (Ed.), *Handbook of Cognitive Neuropsychology*, vol. 9. Amsterdam: Elsevier.

Shafto, M.G., & Remington, R.W. (1990). Building human interfaces to fault diagnostic expert systems II: Interface development for a complex, real-time system. Seattle, WA: In *CHI'90 Workshop on Computer-Human Interaction in Aerospace Systems*.

Shafto, M.G., Remington, R.W., & Trimble, J.P. (1994). *A refinement framework for cognitive engineering in the real world*. Paper prepared for the symposium "Cognitive Science Meets Cognitive Engineering," Cognitive Science Society Annual Conference. Atlanta, GA: Georgia Institute of Technology.

Simon, H. (1991) Cognitive Architectures and Rational Analysis: Comment. In *Architectures for Intelligence / the Twenty-second Carnegie Symposium on Cognition*. Edited by VanLehn, Kurt, Hillsdale, NJ: Lawrence Earlbaum Associates.

Simmons, R. (1994) Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*. 10(1).

Smith, S. and Mosier J. (1986) Guidelines for Designing User Interface Software. Report No. 7 MTR-10090, ESD-TR-86-278. MITRE Corporation.

Stefik, M.J. (1980) *Planning with Constraints*. Report number 80-784 (doctoral dissertation), Computer Science Department, Stanford University.

Stein, Earl S. and Garland, Daniel. (1993) Air traffic controller working memory: considerations in air traffic control tactical operations. FAA technical report DOT/FAA/CT-TN93/37.

Stiles, W.S. (1946) A Modified Hemholtz Line Element in Brightness Colour Space. *Proceedings of the Physical Society of London*, 58, 41-65.

Talotta, Nicholas J. (1992) Controller Evaluation of Initial Data Link Terminal Air Traffic Control Services: Mini Study 2. Volume 1. FAA Technical Report DOT/FAA/CT-92/2,1.

Triesman, A. and Gelade, G. (1980) A Feature Integration Theory of Attention. *Cognitive Psychology*, 12, 97-136.

Triesman, A. and Sato, S. (1990) Conjunction Search Revisited. *Journal of Experimental Psychology: Human Perception and Performance*, 16 (3) 459-478.

Tversky, A. and Kahneman, D. (1974) Judgement Under Uncertainty: Heuristics and Biases. *Science*, 185, 1124-1131.

Van Lehn, K. (1990) Mind Bugs: The Origins of Procedural *Misconceptions. Cambridge, MA: MIT Press*.

Vortac, O.U., Edwards, M.B., Fuller, D.K., and Manning, C.A. (1993) Automation and Cognition in Air Traffic Control. *Applied Cognitive Psychology*, 7 631-651.

Wharton, C., Bradford, J, Jefferies, R., and Franzke, M. (1992) Applying Cognitive Walkthroughs to More Complex User Interfaces: Experiences, Issues, and Recommendations. *Proceedings CHI'92*, ACM, 381-388.

Wolfe, J.M. (1998) What Can One Million Trials Tell Us About Visual Search? *Psychological Science*, 9 (1), 33-39.

Wolfe, J.M. (1994) Guided Search 2.0: A Revised Model of Visual Search. *Psychonomics Bulletin and Review*, 1 (2), 202-238.

Wyszecki, G. and Stiles, W.S. (1967) *Color Science: Concepts and Methods, Quantitative Data and Formulas*. John Wiley and Sons, New York, 1967.