Ritter, F. E., & Major, N. P. (1995). Useful mechanisms for developing simulations for cognitive models. *AI and Simulation of Behaviour Quarterly, 91*(Spring), 7-18.

Useful Mechanisms for Developing Simulations for Cognitive Models

Frank E. Ritter

Nigel P. Major

ESRC Centre for Research in

Development, Instruction and Training

Department of Psychology, University of Nottingham

Nottingham, NG7 2RD, UK

Email: ritter, nigel @psyc.nott.ac.uk

Phone: ++44 (115) 951 5302; Fax: 951 5324

Abstract

We describe an approach for developing cognitive models that can interact with real-time interactive tasks (like flying a plane) and static, puzzle-like tasks. The primary requirement is the ability to create simulations usable by both subjects and models. Additional general simulation requirements can be noted for our applications in particular, such as visual displays. We also consider desirable model environment features. After a description of our sample tasks, and a survey of near misses, we defend our choices of (a) using Garnet as a simulation language, (b) developing a Soar model under Unix and the Soar Development Environment, and (c) the use of sockets as a way to hook them together.

Acknowledgements

We thank Martin Arnold, Ellen Bass, Mike Hucka, Howard Martin, Joe Mertz, Brad Myers, Sarah Nichols, Roberto Ong, Karl Schwamb, Bruce Sherwood, and David Steier for comments and useful discussions. Any errors remain the fault of the authors, and an acknowledgement here does not indicate an endorsement. Support has been provided by the DRA, contract 2024/004, and by the ESRC Centre for Research on Development, Instruction and Training.

Introduction

A general need when creating cognitive models is providing a simulation for them to manipulate. These models need a simulation to interact with and learn from, and subject behaviour to build and test the model acquired from the same simulation or an equivalent.

Previous attempts have used a variety of strategies. Some simulations have provided the model with a set of fixed inputs that require the model to do the same action as the subject did (Peck & John, 1992). Others have implemented the task, such as the Tower of Hanoi, in the modelling language itself (Ruiz & Newell, 1989). More ambitious work has provided the cognitive model with direct access to a simulation, often (but not always) by working within an integrated environment like the Lisp Machine. This has been done in the Tacair-Soar system (Rosenbloom, Johnson, Jones, Koss, Laird, Lehman, et al., 1994). This is an advanced simulation, both from the standpoint of including multiple aircraft, the scope of the cognitive model, and the technologies that have been developed to support the creation and integration of its parts. But the scale of this effort (perhaps 60 months of effort) emphasises that providing this access is not yet routine.

Figure 1 shows the possible relationships between a cognitive model and a simulation. In Figure 1a, two simulations must be created, one to collect data from subjects and one for the model to manipulate. (The simulation for the subjects might not be a simulation, but a real world task realised in hardware instead of software.) This approach increases the required work and questions can be raised about the equivalence of the two simulations. Figure 1b is a better situation, where the model's and the subjects' view of the task are equivalent, limited only by theories of perception. This approach is an important and necessary step if one is interested in creating the situation in Figure 1c, where the cognitive model assists the subject. We are interested in creating an environment that provides these later two situations. We review here the various system components that we are assembling to support creating the situation in Figure 1b and 1c for models of two tasks, which we will use as example tasks.

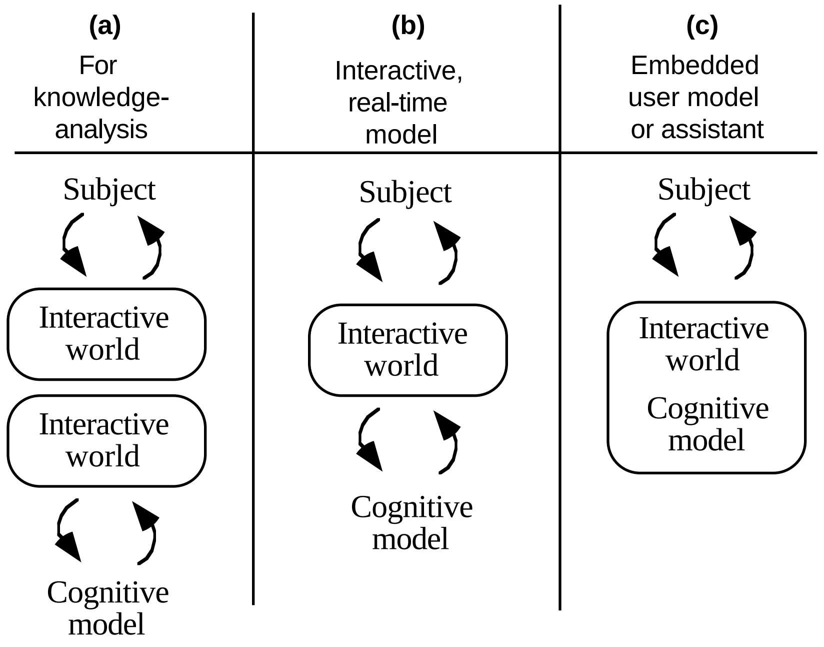

Figure 1: Possible relationships between models and simulations.

Shared features of these (and other) tasks

We have started to model human problem solving in two areas that involve extensive interaction. One is a blocks world type task (building the Tower of Nottingham, a pyramid made out of 21 blocks), the other is an attempt to model real-time problem solving and the development of rapid decision making in a simple Air Traffic Control (ATC) task. In each case, it is not possible to provide the cognitive model[1], in our case, Soar (Newell, 1990; Ritter & Young, 1994; Rosenbloom, Laird, & Newell, 1992),with a simulation written solely in Soar productions, or by providing a fixed set of inputs provided at set times. These tasks have individual requirements, noted below, and several shared requirements, shown in Table 1, that will be common across most tasks. These requirements form a foundation for choosing an approach to creating simulations.

In many cases the simulation tasks are likely to evolve. The simulations are not initially well specified, and will develop over time. Once the model has been created and initially tested, it will also suggest modifications, either to explore additional behaviour or as a way to explore the effect of the world on the task solution.

These sorts of changes can be anticipated for both of our example tasks. The initial model of solving the Tower of Nottingham is likely to suggest additional features of the blocks and their relationships that are necessary to include as perceptual input (we will start with a simplified view). The simple Air Traffic Control-like task is also not completely specified. We foresee additional modification based on the question of how to facilitate the task, either through modifying the display or through the basic support the display provides.

Figure 2a shows how the tower can be built out of blocks that fit together to create a ziggurat (stepped pyramid). Figure 2b shows how two pieces fit together to create a half-level. Two half-levels make a level, and five levels make the tower along with a capping piece. An extensive set of regularities of childrens' behaviour is available, including while they were taught how to solve it by teachers (Wood, Bruner, & Ross, 1976), other children, and a computer tutor (Wood, Shadbolt, Reichgelt, Wood, & Paskiewitz, 1992).

This task makes it clear why the analyst must also see what the model and subject see, including a visual display of what the model could see and what it actually attends to. The position and orientation of the blocks are crucial in performing this task. This information is complicated, and the model's behaviour could vary wildly with small changes.

We are also interested in studying the genesis of rapid decision making (Ong & Ritter, 1995), and have chosen to start with a simple ATC-like task. Because we are free to make changes to suit our needs without regard to external validity our task will differ from real ATC in numerous ways, although some of the results are likely to transfer. For example, real ATC uses paper flight strips to keep track of planes and the big, circular displays one expects are used as a relatively static, current display. In our task, we only expect to have the circular display, which will be more dynamic.

With this task we are definitely not trying to duplicate ATC, instead we are accepting a simplified version so that we can look at the following questions:

- How do people learn to do a real-time process control task?
- What skills and background knowledge do they need to start the task?

[1] In order to avoid confusion and for brevity, we define *the model* to refer to the cognitive model, and *the simulation* to refer to the simulation of the world that the model uses.

Table 1: Requirements for simulations to be used with cognitive modelling.

- What the subject sees should be visible to the model. This implies a theory of visual processing that translates the simulation's objects and provides them to the model. What the model sees should also be visible to the analyst.
- The simulation must be modifiable. Initial versions of the simulation and of the model's perception may be simplistic, but in these tasks we expect the visual format, the task operations, and their perception to play a larger role over time, so the simulation and its interface to the model must be expandable.
- The simulation's actions must be recorded. A basic trace is required showing what the subject and model saw (or could have seen). A complete trace, which would be awkward to use routinely, must also be available for detailed analysis. This need not be stored (although this is safer in case the simulation changes), but may be derivable from a listing of key events.
- Subjects' actions and their times must be recorded. Most cognitive models do not provide predictions with more than 100 ms accuracy, so the actions' timestamps need not be more accurate than 100 ms.
- All controls must be manipulable by the model and subject. In order to explore computer-supported work as in Figure 1c, it is necessary to allow subjects and the model to interact concurrently with the simulation.

Table 2: Environment requirements for studying real-time decision making.

- It must include time stress, requiring decisions to be made rapidly from several options. This translates into 10-50 planes per 10 min. scenario. These numbers are an order of magnitude higher than real ATC, but the amount of handling is relatively small. No negotiation with the aircraft or other control sectors is expected.
- A visual display for subjects and analysts — subjects, so that they can solve the task, and analysts, so that they can understand what the model sees, its behaviour, and internal state.
- A dynamic environment must be provided. Monitoring is an essential aspect of this task, so an environment must be provided with changes not all model initiated.
- Perceptual features of the display must be modifiable and provided to the model. The perceptual ease with which subjects can recognise objects and obtain information is likely to influence how the task is performed.
- The task should force subjects to make errors. Providing an additional type of data to understand behaviour and its evolution.

- How do they use and scan displays?
- How tightly can a model be fitted to human data in these areas? (learning, reaction times, and sequential behaviour)

Table 2 notes several necessary features of such a task to study the evolution of rapid decision making.

Our approach with this paper

We outline here the existing simulation software that is most likely to meet these requirements.

We start with our preference for a particular environment for developing Soar models, perhaps the most difficult aspect of this task, and point out the useful features that should be provided by model development environments. This influences the choice of machine and thus the available simulations and the communication medium between the model and the simulation.

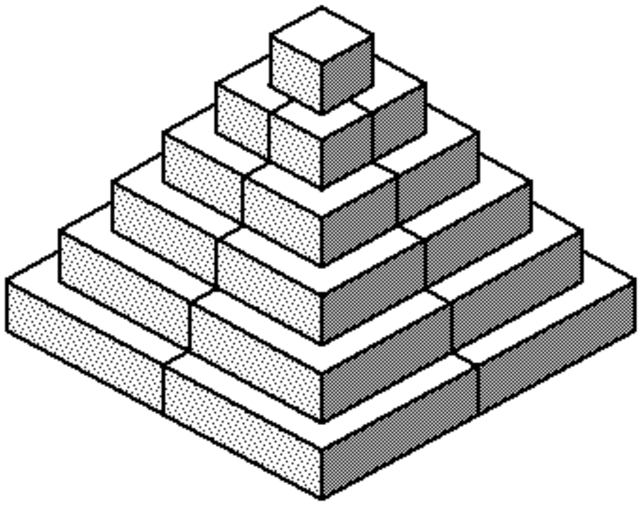

Figure 2a: The tower of Nottingham.

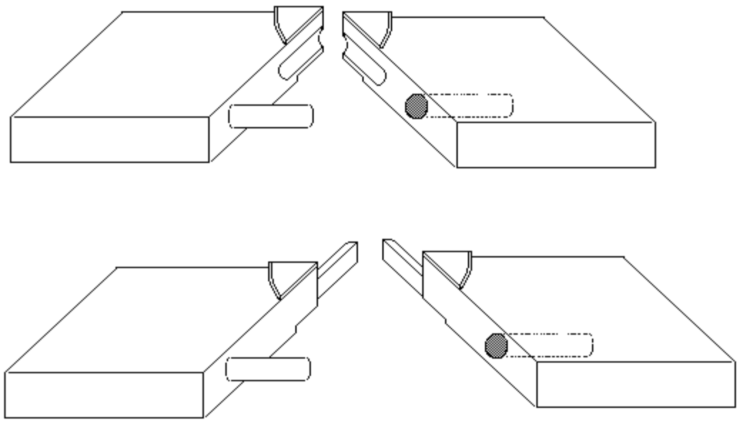

Figure 2b: The pieces that make up each level in the Tower of Nottingham. (Figure courtesy of H. Wood.)

Model development environments

The environment for developing the model is an important decision, perhaps the most important. The analyst needs to be able to see the model, understand how it works, and modify it. We base our discussion around cognitive models in Soar. While some may be directly interested in Soar, this discussion should be seen primarily as a discussion of the types of interfaces that can be provided for cognitive models.

The distributed Soar system provides a command line interface, and runs (after appropriate compilation) on Macintoshes, PCs and Unix machines. These versions are available from the archive noted in the references, and tend to be named based on their release number. Aliases (Nichols & Ritter, in press) are included in the latest Soar release (6.2.5), which offers some customisability.

There are large differences between the existing environments for developing Soar models. The development environment for Soar under Unix is good enough to force the model to be developed there and thus the simulation. Soar and the environments described here are all in the public domain, so price does not affect the choice.

Integrated, structured editors

SDE (Hucka, 1994) is currently the best environment for manipulating Soar models. As a menu-driven editor integrated with the running Soar process, it recognises productions and allows the modeller to manipulate them in a natural way, doing such things as directly loading or tracing them. It also includes some useful displays and commands for finding and structuring the model. It is similar in many ways to the interface provided by Clips. It is a set of modifications to GNU Emacs, similar to and vaguely evolved from earlier efforts with a (relatively) long history (Ritter, Hucka, & McGinnis, 1992). It is still being developed, and users can modify or extend it as well.

It requires Emacs 19.23+ and Emacs' ability to manipulate processes, which is currently available only under the Unix or Linux operating systems. SDE is available from the centro archive.

An interface to Soar based under Microsoft Windows 3.x has also been developed (McMullan & Steier, 1994). It includes a production editor, dialogue boxes for setting various parameters, context-sensitive help and several other features. However, it is not as complete an environment as the SDE, and its top speed on the fastest PCs does not match the speed of the fastest Unix machines. WinSoar is made available to the Soar community on an unsupported basis, but users with only a PC should find it quite valuable and adequate for the development of small-to moderate size models. The latest version can be retrieved from the archive.

Monitoring Displays

The Soar group at USC/ISI have created several displays of working memory elements using raw graphic system calls. These displays have not been generally made available, but are likely to end up in the SDE or the general Soar release. These include "monitor" windows that show the goal stack as the model runs These indicate that those who are sufficiently familiar with Soar, C, and X windows can create their own displays, and that such displays are useful.

For a brief period in 1992, the DSI (Ritter & Larkin, 1994, in press) was the most complete and used interface for Soar (if you counted use of its components). It is now unused because it ran only in the Lisp based version of Soar (Soar5), but it does offers several lessons for modelling. Its graphic displays of the internal state of Soar showed how useful such displays can be for debugging and teaching (e.g., it was used at demos and to help create the Soar video (Lehman, Newell, Newell, Altmann, Ritter, & McGinnis, 1994)). Its functionality is slowly being taken up by the SDE and the monitoring windows. Its only major drawback was that it slowed down Soar by approximately a factor of two. This was caused by two things: (a) It was implemented in an older, larger version of Garnet that made the image large enough to swap, and (b) it included a continuous graphic trace. Sadly, any graphic display running on the same processor as a model will slow it down.

Table 3: Some of the features of TclSoar.

- Full command language including variables, data structures, command structures and file operations.
- Tk extension enables interactive Graphical User Interface (GUI) creation, and a simple process communication mechanism called "send". GUIs can be created and modified without C coding or recompilation. Data from Soar can be sent to other applications (running on other machines, if desired), such as a graphing utility, to generate run-time performance displays.
- Script execution from productions. Hence, a model can call any user command. This enables users to run Tcl/Tk scripts when a particular Soar event occurs.
- Ability to add or communicate with numerous extensions (the Tcl archive contains over 250 extensions). These extensions include graphing utilities, a socket interface, and a hypertext system.
- Portable to Unix, Mac (running a Unix OS), and PC using existing Tcl interpreters.
- More elaborate and context dependent aliases.

Interpreted language with libraries of extension modules

TclSoar (Schwamb, 1994), described in Table 3, is a version of Soar incorporating a Tcl, an embeddable, extendible command line interpreter. TclSoar also includes the Tk extension of widgets (graphical objects) that enables users to create graphical extensions interactively. To find out more about TclSoar, Tcl and Tk see the web documentation: http://www.isi.edu/soar/schwamb. TclSoar is available from the archive.

In addition to providing a more extendible command line interface, TclSoar makes it easier to connect graphical interactors and simulations to Soar. It provides access to a simple simulation language, existing Tcl simulations, and an additional model/simulation interprocess control approach, While it will be a master at none of these, its greatest potential is serving as a uniform and integrating environment for incorporating additional displays of the running model, creating simple simulations easily, and hooking up more complicated simulations routinely.

Using existing simulations

We outline several existing simulations to further define our needs and review possibilities that others may wish to consider. Modellers may be able to find other simulations related to their needs. Table 4 lists several of the advantages of using an existing simulation. Despite these advantages, few simulations meet the needs listed in Table 2; communication between the model and the simulation typically is a problem, particularly extending the simulation to provide reasonable perception to the model.

Simulators at Brooks AFB

Psychology labs studying learning or performance will often have existing tasks that they use. For example, there are several simulation development efforts going on at Armstrong Lab at Brooks Air Force Base to support investigations into individual differences, intelligent tutoring systems, and rapid decision making (Regian & Shute, 1993). As part of their work on studying automated instruction, the group has developed 14 criterion tasks. A typical example is Space Fortress, a well studied task (see Vol. 71 of Acta Psychologica, 1989). It is essentially an asteroids-like video game. These simulations do not appear to be appropriate for the initial simulation of studying rapid decision making (or children's development), but are likely to be useful as an additional task for studying rapid decision making.

TRACON simulations

Simulations may also be available commercially. Wesson International of Austin, TX, (ph. 512-328-0100) offers a range of ATC simulators called TRACON. The simplest version costs approximately US$70 and runs on the PC or Macintosh. New scenarios can be added, but it cannot be otherwise modified, instrumented, or run by other software. A complete version is available for approximately $25,000 that runs on dedicated hardware. This version is used at the (US) Federal Aviation Agency Academy to train tower operators and for research.

SGI flight simulator

Some simulators come bundled with the operating system or the hardware. For example, there is a flight simulator available that runs on and takes advantage of the special hardware features of Silicon Graphics computers. While it is not completely appropriate for our simulation, it is interesting because a Soar system has been created that flies it (Pearson, Huffman, Willis, Laird, & Jones, 1993).

General purpose simulation languages

If an appropriate simulation is not available, one must be built. This can be done in a standard programming language, but special simulation and interface languages and toolkits can ease the task of creating a simulation that will run on multiple machines.

We currently prefer the Garnet system and have started an effort to create a simple ATC simulation in it that will meet the requirements noted in Tables 1 and 2 (Ong & Ritter, 1995). The simulation will need to run on what is natural for the model (this will be Unix for Soar) and what is natural for the subject (this will be a system that can provide accurate timing information, such as the Macintosh or PC).

There are other platform independent toolkits for creating graphical interfaces, so we include a sample of the most likely candidates. What we do not cover are environments that our modelling language does not have specific affinity for, that we don't have access to, or that we cannot add easily to our environment.

Table 4: Possible advantages to using an existing simulation.

- Savings in development time
- Community of existing users
- Possible existing analyses and summaries of behaviour
- Possible existing models of behaviour for comparison

Garnet

Garnet (Myers, et al. 1990) calls itself a 'graphical user interface management system', but it is also an object oriented graphical interface toolkit and environment. It is currently written in Common Lisp, and runs on Unix and the Macintosh. It is actively being developed at CMU by the Garnet group. While bug responses are good, there is lots of help because it comes with source code and there is a Usenet bulletin board (comp.windows.garnet) for users.

Garnet is available via anonymous FTP from a.gp.cs.cmu.edu (128.2.242.7). You must directly change to the distribution directory with "cd /usr/garnet/garnet". Get the README file there and follow its directions.

A C++-based Garnet is under development, but it is not clear the implications of this, as Garnet takes advantage of many Lisp features, including its interpreter. This version will be developed to be portable across Unix, Macs and PCs. It is to be as easy to learn as SUIT (noted below) but capable of scaling up to full-size interfaces. There will be some kind of interpreted language eventually, similar to function to Tcl, such as Scheme, interpreted C, or both to describe an object's behaviour rather than a new language.

cT

cT (Sherwood & Andersen, 1993) is a Pascal-like language developed by people sympathetic to tutoring systems. Modellers interested in simpler simulations and who don't already know Lisp, would find this a particularly attractive language to work with. It includes a visual display and simple user interface components such as menus. Source code (including displays) can run on Unix machines, Macs and PCs. It has been used to provide several Soar models with simulations using sockets on Unix systems, and can provide the same inter-process communication ability (pseudo-sockets) across networked Macintoshes.

cT becomes unsuitable for large simulations (such as the ATC task) because it is a relatively low level language. cT supports multi-dimensional arrays with bounds checking, for example, but no data or record structures. There will soon be a version 2.5, with significant enhancements for multimedia such as support for QuickTime and Video for Windows.

The Unix version is freely available via anonymous FTP from columbus.andrew.cmu.edu in the directory "/pub/ct". The DOS and Macintosh versions are available from Physics Academic Software (PO Box 8202, North Carolina State University, Raleigh, NC 27695; tel: 919-515-7447; fax: 919-515-2682) for $175 (or $250 for both DOS and Mac). There is a users' group, ctug@athena.mit.edu.

GNU Emacs Lisp

GNU Emacs has a rather open architecture, including a Lisp extension language. It would be straightforward to create a simulation in GNU Emacs Lisp and tie it to a model using the process control in SDE (Hucka, 1994). This would not provide a fully graphical display as we need for our tasks and the size of the simulation would be limited by the speed of this Lisp, but it would allow a model to talk to a piece of software with a textual interface, such as the Dismal spreadsheet (Nichols & Ritter, in press).

Model-based simulations

It is often possible to create a simulation in the modelling language itself. In Soar, these range from productions directly implementing the task to more sophisticated systems for accessing external timers and files (Guttman & Huffman, 1992; Nelson, 1994). These approaches are useful for fixed sets of inputs, and simple interactions (such as the Tower of Hanoi), but they are not powerful enough to implement either of our tasks. This approach can be awkward because modelling languages are not designed to be simulation languages. The approaches one must take are often not supported by debuggers and visual displays because they use the language in odd ways, and it is typically not possible to create a display that subjects can interact with.

Raw Xlib Window calls in C

It will often be possible to implement a simulation in the language used to implement the model. Simulations created in the foundation language are easy to connect to the model. For Soar it is C. It is possible to use C and raw calls to the Xlib window functions to create a simulation and graphic display under the X window system (Jones, 1989). Schwamb at ISI (personal communication, 1994) recommends this approach where speed is an issue, for it produces the fastest simulation because it is linked into the native display system. It will perhaps be the slowest way to develop because little support is provided, and it requires the most skill to implement.

Other simulation languages

A wide listing of platform independent graphical user interface building tools is available as a frequently asked questions (FAQ) posting to comp.windows.misc (also news.answers) by Guthrie (wade@nb.rockwell.com). This FAQ and ones on Lisp and window systems often posted to comp.lang.lisp were used to help derive the following other candidates.

Visual Basic. Visual Basic (Anderson, 1994) is a user interface rapid prototyping language that comes with widgets that can have structured BASIC code attached. It is possible to create good interfaces fairly quickly. The downside is that the code is hard to manage. Visual Basic is not free, but not expensive, a few hundred pounds.

SUIT. Simple User Interface Toolkit (SUIT) is a C-based system developed at the U. of Virginia. It promises to be easy to use ("up and running simple applications in 2 hours (estimated time)"). It is freely available via FTP and is actively maintained. If the Garnet based system is scrapped, we are likely to take this system very seriously. For more information, finger suit@uvacs.cs.virginia.edu.

Winterp. WINTERP is a object-oriented toolkit developed in XLISP (a public domain Lisp). It provides a number of advanced features coupled with the flexibility and power of a small efficient Lisp language and object system. It is available via anonymous FTP at ftp.x.org in “/contrib/devel_tools”.

Model - simulation communication channels

As the model runs it must query the simulation for input and provide the simulation with task commands. Table 5 provides a summary of the features of various communication channels for tying models to simulations. For us, a communication medium’s ease of use, flexibility, and robustness under use is more important than its bandwidth. The amount of information in tasks pass between models and their simulations can be provided by most communication media. At present, the input from the simulation solely represents the result of visual scanning of the simulation's displays, but we anticipate incorporating a verbal input as well. When presented symbolically, we expect that this will represent no more than 100 symbols per second. At present, the output from the model will consist solely of keystrokes and mouse movements and clicks, although simulated verbal output may be added later. The output will also be symbolised, and we expect that it will not exceed 10 symbols per second. We expect to use sockets in both of our actual tasks, but may briefly explore the feasibility of using Emacs processes as well.

There are other methodologies and mechanisms for interprocess communication. Stevens

Table 5: Features of various communication channels.

	Bandwidth	**Ease of creation**	**Flexibility**	**Robustness**
(a) Plain files	Low	Good	Bad	OK
(b) Sockets	High	OK	OK	Good
(c) Emacs processes	Medium	OK	Good	Maybe bad
(d) Joint Compilation	V. High	Bad	OK	V. Good
(e) Apple events	Medium	Bad	Bad	OK

(1990) provides more Unix possibilities. We only offer the most likely candidates, particularly ones that can run between and on multiple platforms, or on the platforms we will have to use.

Plain file reading and writing

The simplest and quickest method of tying two processes together is to have one write to a disk file and the other to read it. This approach gets complicated when multiple messages are passed and the read/write cycles become often enough that the two processes contend for the same file (if they don't actually accidentally overwrite it), although there are semaphore techniques available to avoid clashes. This approach thus suffers from low bandwidth and complete reliance on the operating system to adjudicate reads and writes. We do not recommend it, but introduce it first as it is a precursor to Unix sockets, which are similar but more robust.

Sockets

Sockets are a well-used Unix mechanism (including AUX on the Macintosh) for allowing two processes to communicate. (Further information is readily available in textbooks such as Glass, 1993.) They are similar to files in how processes read and write them, but they do not have disk space allocated (actually, when a disk file is manipulated this is also done with a socket, but this is transparent to the user). Sockets are more costly to set up in programming time than we currently would like (they are a relatively low level communications medium), but relatively robust and portable, and can run across and between machines. They offer a relatively high process communication bandwidth. Starting with code generated by Mertz and Pelton, we have created a package (MONGSU, available in the Soar archive) that makes using this channel simpler (Ong & Ritter, 1995). While it was designed for Soar and Garnet, any combination of C and Lisp systems that need to communicate may benefit.

GNU Emacs

GNU Emacs runs on Unix, VMS, and DOS platforms. There are copies emerging for the Mac, but they are of no use here because they do not yet incorporate process control (such as running a model). Emacs offers a relatively simple, lightweight approach to tying processes together. When a process spawned within Emacs returns a value, a set of patterns are used to assign an associated Lisp function to handle the output. Typically this is simply to print out the text, but this also offers a simple way to pass messages between two processes. It could be used to tie the model quickly to a simulation.

This would offer several advantages if it were available as a utility, which it might be possible to build fairly easily. This system would be flexible because it is implemented in an interpreted language that includes a debugger. It could be extended easily to pass information to several processes at once or to record messages for later analyses. Experience with SDE (Hucka, 1994) suggests that the bandwidth of this approach is "Good".

This approach has one main disadvantage. This mechanism is less robust because it is possible for the user to interrupt the handler (perhaps by mistake).

Microsoft Windows interprocess communication

Dynamic Link Libraries (DLL) provide a means of sharing functions between different Microsoft Windows applications. These functions have to be linked in, but are fairly straightforward for those who know them. They tend to be used for side-effects and not for passing information. Dynamic data exchange (DDE) is a system for automatically passing data between Windows applications.

DLL is similar to joint compilation, and DDE similar to sockets. Both of these essentially represent joint compilation in an environment designed to support this approach (Microsoft wrote both Windows and the major horizontal applications). This would be a good way to interface Soar to existing software, probably much easier than using Apple Events, but not useful for Soar running under the UNIX environment. Modelling perception of what is on the screen would still be difficult to add if applications did not include it.

Apple events

Apple events are objects/messages that can be sent to most programs running on the Macintosh. These events correspond to user input, such as keystrokes and mouse movements, or pre-specified queries about contents. There are three problems with using this. First, while it is intrinsic to the operating system, relatively less work has gone into developing 'reading' events, such as "What's in location (2,3)?". ‘Reading’ events require a separate function to be written and linked into the simulation. This is not different from a socket handler, but the technology of developing such programs appears to be less mature. Second, finding expertise in this area, in people and documentation, appears to be difficult, making its use more difficult. Third, there are rumours that it will not be supported in later versions of the Macintosh operating systems.

Joint compilation/linking

The fastest and most robust way to link two processes together is to compile them together and have them call each other. This approach is used to link Soar and the ModSAF airplane simulation environment (Schwamb, Koss, & Keirsey, 1994). The advantages of this approach are that whole structures can be passed (as they are not actually separate processes), and that the communication rate is as fast as possible.

We believe that the disadvantages of joint compilation will outweigh the advantages for most users. The mutual calling of functions requires the simulation (for Soar) to be written in C, and duplicates what are normally operating system functions. Changes to one of the systems requires knowledge of the other's internal functions and data structures, and relinking or recompiling. Because there is no intermediary, observing and thus debugging this link may be difficult.

Hardware platforms

The requirements and suggested solutions in the previous sections for developing the cognitive model, developing the simulation, and the process communication channel to use, directly influence the choice of computer. These needs split into two set of requirements met by different machines. We will need to use two systems, one for developing the model and one for running subjects.

It is worth mentioning an additional timing problem that may occur. Unix and other multi-tasking systems have essentially non-deterministic schedulers. This is, the time and order of running the model and the simulation will vary. On large time scales, there is plenty of time for the display information to be passed to the model and for the model to pass back commands. On short time scales, the model and simulation may not see time pass equally. Multi-agent Soar (Hucka, personal communication, 1994) was created was to get round this non-deterministic scheduling for multiple models by running them synchronously (i.e., so that they each see each model cycle at the same time).

Unix

Currently, machines that routinely run Unix as their operating system are generally the fastest and most commonly available for developing cognitive models and simulations. Their only drawback is that they cannot automatically provide reliable timing information about user's input. As a multitasking operating system, with multiple processes and a scheduler, a user action is not necessarily timestamped when it is input. Delays of 10 ms to several seconds are typical. For example, although time stamps with 200 ms accuracy are quite likely with an available GNU Emacs package, log.el by Erik Altmann incorporated into Dismal, a significant portion of the keypresses, perhaps 5%, take 0 ms.

It is not possible to have a single user process running under the Unix operating system in a routine or simple manner. However, timing glitches are minimised if the simulation process runs at maximum priority, timestamps incoming keystrokes first, enough swap space is provided, and some other minor resource issues are resolved. Another option is having a small, high-priority process that just received keystroke data, timestamped it, and passed to the simulation. There also are real-time Unix kernels that provide pre-emptive scheduling, but these are rare and require replacing the standard operating system. Of course, with a PC or Mac, there are no other process running so timing (done right) is less of an issue.

Logs can be created from videotaping the users, but this is expensive. So while machines running Unix are likely to be the machine of choice for developing models and simulations, another machine running the simulation must be used to record the subject's data.

Micro-computers

Micro-computers can provide accurate timing information. If platform independent software is used to develop the simulation, subjects can be run on both platforms. The drawbacks that preclude the use of Macintoshes as a complete environment include the lack of model development environment and the relative difficulty of inter-process communication.

PC equivalents (386 or better) can provide acceptable environments for developing models. They appear to support more stand-alone simulations than the other hardware platforms combined. What they lack is a superior model development environment, a complete simulation development environment, and a robust and easy to use communication medium.

We expect that the PowerMacintosh, a cross between a Macintosh and a PC, will become important for developing systems. It is a faster and has a more open architecture. Recent press releases from Apple and Digitool suggest that MacLisp will be ported to the PowerPC's native code.

Recommendations

We end up with two recommendations. The first, which you could do now, is to use the existing software and systems recommended above. The analyst will prefer to use (or extend) an existing simulation if one is available that meets their criteria. In many cases it appears that simulations have not been built with cognitive modelling in mind, so this will not be possible and a simulation will have to be built. There is a range of simulation building tools available, starting with Soar-Sim, which is adequate for simple models that simply process set inputs. Slightly more complex simulations can be implemented in languages like cT, which is easy to work with. More complicated simulations requiring more powerful toolkits will find Garnet useful. Most analysts will wish to develop the model using the a structured, integrated editor, which currently requires a Unix machine (for Soar at least). Sockets will be the communication medium of choice using the MONGSU library.

The second recommendation is to include an interpreted extension language. Garnet provides this. When complete, TclSoar will show that the Tcl can serve as well.

We have started developing both the ATC task (Ong & Ritter, 1995) and the Tower of Nottingham task with Garnet. The models and simulations run on a Unix machine, communicating using sockets. The subjects' version of the ATC simulation runs on the Macintosh. We will not immediately create a version of the Tower of Nottingham for running subjects because we believe that we have adequate data at this point, and because validating the simulation as comparable to real blocks could be quite difficult.

References

An archive of Soar software is available via anonymous FTP from host centro.soar.cs.cmu.edu in the directory "/afs/cs.cmu.edu/project/soar/public/Soar6". If you start to use any of this code, please email soar-requests@cs.cmu.edu to notify them so that they can add you to appropriate mailing lists, provide updates, and recognise you when you send in bug reports.

Anderson, T. (1994, May). Visual Basic for applications. *Personal Computer World*, 492-496.

Glass, G. (1993). *Unix for programmers and users: A complete guide.* Englewood Cliffs, NJ: Prentice-Hall.

Guttman, R. H., & Huffman, S. B. (1992). SoarSim: A Soar simulation-building tool (version 1.1 user's manual). Artificial Intelligence Laboratory, U. of Michigan.

Hucka, M. (1994). The Soar Development Environment. Artificial Intelligence Laboratory, University of Michigan.

Jones, O. (1989). *Introduction to the X Window System*. Englewood Cliffs, NJ: Prentice-Hall.

Lehman, J. F., Newell, A., Newell, P., Altmann, E., Ritter, F., & McGinnis, T. (1994). The Soar video. The Soar Group.

McMullan, J., & Steier, D. (1994). WinSoar: Soar for IBM PC-compatibles running MS-Windows. In T. Johnson (Ed.), *Soar Workshop 13*. 164-165.

Myers, B. A., Giuse, D. A., Dannenberg, R. B., Vander Zanden, V., Kosbie, D. S., Pervin, E., Mickish, A., & Marchal, P. (1990). Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer, 23* (11), 71-85.

Nelson, G. (1994). SimTime User's Guide. FTPable from centro.soar.cs.cmu.edu: "/afs/cs/project/soar/public/Soar6/user-library/SimTime.tar.Z". The Soar Group, School of Computer Science, Carnegie-Mellon University.

Newell, A. (1990). *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.

Nichols, S., & Ritter, F. E. (in press). A theoretically motivated tool for automatically generating command aliases. In *Proceedings of CHI '95*.

Ong, R., & Ritter, F. E. (1995). Mechanisms for routinely tying cognitive models to interactive simulations. In *HCI International '95*. Osaka, Japan:

Pearson, D. J., Huffman, S. B., Willis, M. B., Laird, J. E., & Jones, R. M. (1993). A symbolic solution to intelligent real-time control. *Robotics and Autonomous Systems, 11* , 279-291.

Peck, V. A., & John, B. E. (1992). Browser-Soar: A computational model of a highly interactive task. In *CHI'92 - Conference on Human Factors and Computing Systems*. 165-172. New York: ACM Press.

Regian, J. W., & Shute, V. J. (1993). Basic research on the pedagogy of automated instruction. In D. M. Towne, T. de Jong, & H. Spada (Eds.), *Simulation-based experiential learning*. (pp. 121-132). Berlin: Springer-Verlag.

Ritter, F. E., Hucka, M., & McGinnis, T. F. (1992). Soar-mode Manual. CMU-CS-92-205. School of Computer Science, Carnegie-Mellon University.

Ritter, F. E., & Larkin, J. H. (1994, in press). Using process models to summarize sequences of human actions. *Human-Computer Interaction* .

Ritter, F. E., & Young, R. M. (1994). Practical introduction to the Soar cognitive architecture: Tutorial report. *AISB Quarterly, 88* , 62.

Rosenbloom, P. S., Johnson, W. L., Jones, R. M., Koss, F., Laird, J. E., Lehman, J. F., Rubinoff, R., Schwamb, K. B., & Tambe, M. (1994). Intelligent automated agents for tactical air simulation: A progress report. In *Proceedings of the Fourth Conference on computer generated forces and behavioral representation*. Gainsville: U. of Central Florida.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (1992). *The Soar papers: Research on integrated intelligence*. Cambridge, MA: MIT Press.

Ruiz, D., & Newell, A. (1989). Tower-noticing triggers strategy-change in the Tower of Hanoi: A Soar model. In *The Annual Conference of the Cognitive Science Society*. 522-529.

Schwamb, K. (1994). TclSoar: A programmable Soar shell. In *Soar Workshop XIV*. 73-79. U. of Michigan:

Schwamb, K. B., Koss, F. V., & Keirsey, D. (1994). Working with ModSAF: Interfaces for programs and users. In *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Technical Report IST-TR-94-12: Institute for Simulation and Training.

Sherwood, B. A., & Andersen, D. M. (1993). cT creates prize-winning portable physics programs. *Computers in Physics, 7* (2), 136-143.

Stevens, W. R. (1990). *Unix Network Programming*. Englewood Cliffs, NJ: Prentice-Hall.

Wood, D., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry, 17* , 89-100.

Wood, D., Shadbolt, N., Reichgelt, H., Wood, H., & Paskiewitz, T. (1992). EXPLAIN: Experiments in planning and instruction. *AISB Quarterly, 81* , 13-16.