Advanced Control Mechanisms

Appendix 2

This appendix is concerned with specific control mechanisms that are provided by programming languages or that may be implemented on top of existing languages as aids to doing computer vision. The treatment here is brief; our aim is to expose the reader to several ideas for control of computer programs that have been developed in the artificial intelligence context, and to indicate how they relate to the main computational goals of computer vision.

A2.1 STANDARD CONTROL STRUCTURES

For completeness, we mention the control mechanisms that are provided as a matter of course by conventional research programming languages, such as Pascal, Algol, POP-2, SAIL, and PL/1. The influential language LISP, which provides a base language for many of the most advanced control mechanisms in computer vision, ironically is itself missing (in its pure form) a substantial number of these more standard constructs. Another common language missing some standard control mechanisms is SNOBOL. These standard constructions are so basic to the current conception of a serial von Neumann computer that they are often realized in the instruction set of the machine. In this sense we are almost talking here of computer hardware.

The standard mechanisms are the following:

1. *Sequence.* Advance the program counter to the next intruction.
2. *Branch instruction.* Go to a specific address.
3. *Conditional branch.* Go to a specific address if a condition is true, otherwise, go to the next instruction.
4. *Iteration.* Repeat a sequence of instructions until a condition is met.

5. *Subroutines.* Go to a certain location; execute a set of instructions using a set of supplied parameters; then return to the next instruction after the subroutine call.

All the standard control structures should be in the toolkit of a programmer. They will be used, together with the data structures and data types supplied in the working language, to implement other control mechanisms. The remainder of this appendix deals with "nonstandard" control mechanisms; those not typically provided in commercial programming languages and which have no close correlates in primitive machine instructions. Nonstandard control mechanisms, although not at all domain-specific, have developed to meet needs that are not the "lowest common denominator" of computer programming. They impose their own view of problem decomposition just as do the standard structures.

Less standard mechanisms are *recursion* and *co-routining.* Co-routining can be thought of as a form of recursion.

A2.1.1 Recursion

Recursion obeys all the constraints of subroutining, except that a routine may call upon "itself." The user sees no difference between recursive and nonrecursive subroutines, but internally recursion requires slightly more bookkeeping to be performed in the language software, since typically the hardware of a computer does not extend to managing recursion (although some machines have instructions that are quite useful here).

A typical use of a recursive control paradigm in computer vision might be:

```
To Understand-Scene (X);
(
If Immediately-Apparent(X)
then Report-Understanding-Of(X);
else
   (SimplerParts ← Decompose(X);
   ForEach Part in SimplerParts
      Understand-Scene(Part);
   )
[ )
```

Recursion is an elegant way to specify many important algorithms (such as tree traversals), but in a way it has no conceptual differences from subroutining. A routine is broken up into subroutines (some of which may involve smaller versions of the original task); these are attacked sequentially, and they must finish before they return control to the routine that invokes them.

A2.1.2 Co-Routining

Co-routines are simply programs that can call (invoke) each other. Most high-level languages do not directly provide co-routines, and thus they are a nonstandard control structure. However, co-routining is a fundamental concept [Knuth 1973]

and serves here as a bridge between standard and nonstandard control mechanisms.

Subroutines and their calling programs have a "slave–master" aspect: control is always returned to the master calling program after the subroutine has carried out its job. This mechanism not only leads to efficiencies by reducing the amount of executable code, but is considered to be so useful that it is built into the instruction set of most computers. The pervasiveness of subroutining has subtle effects on the approach to problem decomposition, encouraging a hierarchical subproblem structure. The co-routine relationship is more egalitarian than the subroutine relationship. If co-routine *A* needs the services of co-routine *B*, it can call *B*, and (here is the difference) conversely, *B* can call *A* if *B* needs *A*'s services.

Here is a simple (sounding) problem [Floyd 1979]: "Read lines of text, until a completely blank line is found. Eliminate redundant blanks between the words. Print the text, 30 characters to a line, without breaking words between lines." This problem is hard to program elegantly in most languages because the iterations involved do not nest well (try it!). However, an elegant solution exists if the job is decomposed into three co-routines, calling each other to perform input, formatting, and output of a character stream.

A useful paradigm for problem solving, besides the strictly hierarchical, is that of a "heterarchical" community of experts, each performing a job and when necessary calling on other experts. A heterarchy can be implemented by co-routines. Many of the nonstandard mechanisms discussed below are in the spirit of co-routines.

A2.2 INHERENTLY SEQUENTIAL MECHANISMS

A2.2.1 Automatic Backtracking

The PLANNER language [Hewitt 1972] implicitly implemented the feature of "automatic backtracking." The advisability of uniformly using this technique, which is equivalent to depth-first search, was questioned by those who wished to give the programmer greater freedom to choose which task to activate next [Sussman and McDermott 1972].

A basic backtracking discipline may be provided by recursive calls, in which a return to a higher level is a "backtrack." The features of automatic backtracking are predicated on an ability to save and reinstate the computational state of a process automatically, without explicit specification by the programmer.

Automatic backtracking has its problems. One basic problem occurs in systems that perform inferences while following a particular line of reasoning which may ultimately be unsuccessful. The problem is that along the way, perhaps many perfectly valid and useful computations were performed and many facts were added to the internal model. Mixed in with these, of course, are wrong deductions which ultimately cause the line of reasoning to fail. The problem: After having restored control to a higher decision point after a failure is noticed, how is the system

to know which deductions were valid and which invalid? One expensive way suggested by automatic backtracking is to keep track of all hypotheses that contributed to deriving each fact. Then one can remove all results of failed deduction paths. This is generally the wrong thing to do; modern trends have abandoned the automatic backtracking idea and allow the programmer some control over what is restored upon failure-driven backtracking. Typically, a compromise is implemented in which the programmer may mark certain hypotheses for deletion upon backtracking.

A2.2.2 Context Switching

Context switching is a general term that is used to mean switching of general process state (a control primitive) or switching a data base context (a data access primitive). The two ideas are not independent, because it could be confusing for a process to put itself to sleep and be reawakened in a totally different data context.

Backtracking is one use of general control context switching. The most general capability is a "general GO TO." A regular GO TO allows one to go only to a particular location defined in a static program. After the GO TO, all bindings and returnpoints are still determined by the current state of processing. In contrast, a general GO TO allows a transfer not only across program "space," but through program "time" as well. Just as a regular GO TO can go to a predefined program label, a general GO TO can go to a "tag" which is created to save the entire state of a process. To GO TO such a tag is to go back in time and recreate the local binding, access, control, and process state of the process that made the tag.

A good example of the use of such power is given in a problem-solving program that constructs complex structures of blocks [Fahlman 1974].

A2.3 SEQUENTIAL OR PARALLEL MECHANISMS

Some language constructs explicity designate parallel computing. They may actually reflect a parallel computing environment, but more often they control a simulated version in which several control paths are maintained and multi-processed under system control. Examples here are module and message primitives given below and statements such as the CO-BEGIN, CO-END pairs which can bracket notionally parallel blocks of code in some Algol-like language extensions.

A2.3.1 Modules and Messages

Modules and messages form a useful, versatile control paradigm that is relatively noncommittal. That is, it forces no particular problem decomposition or methodological style on its user, as does a pure subroutine paradigm, for example. Message passing is a general and elegant model of control which can be used to subsume others, such as subroutining, recursion, co-routining, and parallelism [Feldman 1979].

There are many antecedents to the mechanism of modules communicating by messages described here. They include [Feldman and Sproull 1971; Hewitt and

Smith 1975; Goldberg and Kay 1976; Birtwhistle et al. 1973]. In the formulation presented by Hewitt, the message-passing paradigm can be extended down into the lowest level of machine architecture. The construction outlined here [Feldman 1979] is more moderate, since in it the base programming language may be used with its full power, and itself is not module and message based.

A program is made up of *modules.* A module is a piece of code with associated local data. The crucial point is that the internal state of a module (e.g. its data) is not accessible to other modules. Within a module, the base programming language, such as Algol, may be used to its full power (subroutine calls, recursion, iteration, and so forth are allowed). However, modules may not in any sense "call upon" each other. Modules communicate only by means of *messages.* A module may send a message to another module; the message may be a request for service, an informational message, a signal, or whatever. The module to whom the message is sent may, when it is ready, receive the message and process it, and may then itself send messages either to the original module, or indeed to any combination of other modules.

The module–message paradigm has several advantages over subroutine (or co-routine) calls.

1. If subroutines are in different languages, the subroutine call mechanisms must be made compatible.
2. Any sophisticated lockout mechanism for resource access requires the internal coding of queues equivalent to that which a message switcher provides.
3. A subroutine that tries to execute a locked subroutine is unable to proceed with other computation.
4. Having a resource always allocated by a single controlling module greatly simplifies all the common exclusion problems.
5. For inherently distributed resources, message communication is natural. Module-valued slots provide a very flexible but safe discipline for control transfers.

Another view of messages is as a generalization of parameter lists in subroutine or coroutine calls. The idea of explicitly naming parameters is common in assembly languages, where the total number of parameters to a routine may be very large. More important, the message discipline presents to a module a collection of suggested parameters rather than automatically filling in the values of parameters. This leads naturally to the use of semantic checks on the consistency of parameters and to the use of default values for unspecified ones, which can be a substantial improvement on type checking. The use of return messages allows multiple-valued functions; an answer message may have several slots. Messages solve the so-called "uniform reference problem"—one need not be concerned with whether an answer (say an array element) is computed by a procedure or a table.

There is yet another useful view of messages. One can view a message as a partially specified relation (or pattern), with some slot values filled in and some unbound. This is common in relational data bases [Astrahan et al. 1976] and artificial intelligence languages [Bobrow and Raphael 1974]. In this view, a mes-

sage is a task specification with some recipient and some complaint departments to talk to about it. Various modules can attempt to satisfy or contract out parts of the task of filling in the remaining slots. A module may handle messages containing slots unknown to it. This allows several modules to work together on a task while maintaining locality. For example, an executive module could route messages (on the basis of a few slots that it understands) to modules that deal with special aspects of a problem using different slots in the message.

There is no apparent conflict between these varying views of messages. It is too early in their development to be sure, but the combined power of these paradigms seems to provide a qualitative improvement in our ability to develop vision programs.

A2.3.2 Priority Job Queue

In any system of independent processes on a serial computer, there must be a mechanism for scheduling activation. One general mechanism for accomplishing scheduling is the priority job queue. Priority queues are a well-known abstraction [Aho et al. 1974]. Informally, a priority job queue is just an ordered list of processes to be activated. A monitor program is responsible for dequeueing processes and executing them; processes do not give control directly to other processes, but only to the monitor. The only way for a process to initiate another is to enqueue it in the job queue. It is easiest to implement a priority job queue if processes are definable entities in the programming language being used; in other words, programs should be manipulable datatypes. This is possible in LISP and POP-2, for example.

If a process needs another job performed by another process, it enqueues the sub job on the job queue and *suspends* itself (it is *deactivated*, or put to sleep). The sub job, when it is dequeued and executed by the monitor, must explicitly enqueue the "calling" process if a subroutining effect is desired. Thus along with usual arguments telling a job what data to work on, a job queue discipline implies passing of control information.

Job queues are a general implementational technique useful for simulating other types of control mechanisms, such as active knowledge (Chapter 12). Also, a job queue can be used to switch between jobs which are notionally executing in parallel, as is common in multiprocessing systems. In this case sufficient information must be maintained to start the job at arbitrary points in its execution.

An example of a priority job queue is a program [Ballard 1978] that locates ribs in chest radiographs. The program maintains a relational model of the ribcage including geometric and procedural knowledge. Uninstantiated model nodes corresponding to ribs might be called hypotheses that those ribs exist. Associated with each hypothesis is a set of procedures that may, under various conditions, be used to verify it (i.e., to find a rib). Procedures carry information about preconditions that must be true in order that they may be executed, and about how to compute estimates of their utility once executed. These descriptive components allow an executive program to rank the procedures by expected usefulness at a given time.

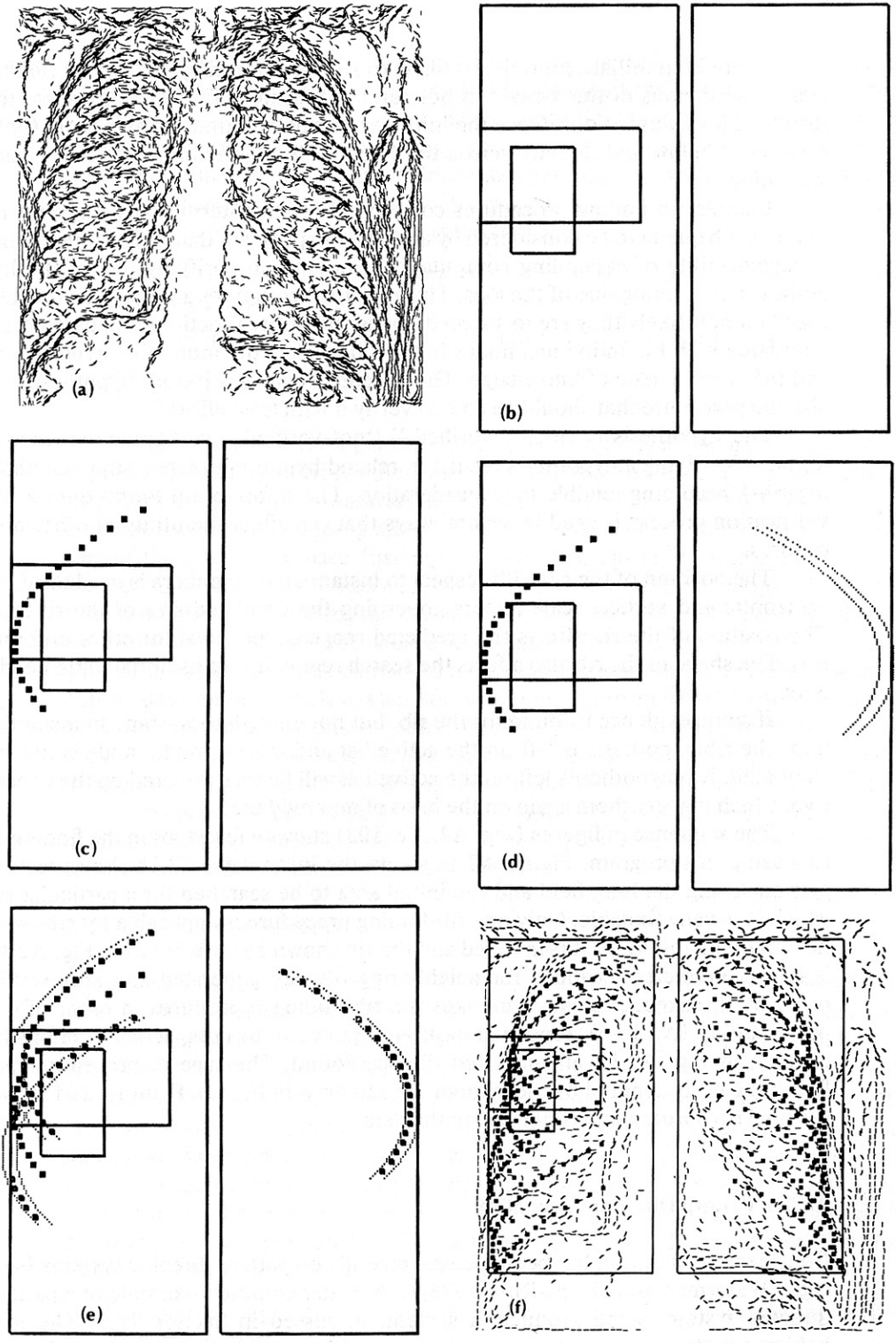

Fig. A2.1 The rib-finding process in action (see text).

There is an initial action that is likely to succeed (locating a particular rib that is usually obvious in the x-ray). In heterarchical fashion, further actions use the results of previous actions. Once the initial rib has been found, its neighbors (both above and below and directly across the body midline) become eligible for consideration.

Eligible rib-finding procedures correspond to short-term plans; they are all put on a job queue to be considered by an *executive* program that must compute the expected utility of expending computational energy on verifying one of the hypotheses by running one of the jobs. The executive computes a priority on the jobs based on how likely they are to succeed, using the utility functions and parameters associated with the individual nodes in the rib model (the individual hypotheses) and the current state of knowledge. The executive not only picks a hypothesis but also the procedure that should be able to verify it with least effort.

The hypothesis is either "verified," "not-verified," or "some evidence is found." Verifying a hypothesis results in related hypotheses (about the neighboring ribs) becoming eligible for consideration. The information found during the verification process is used in several ways that can affect the utility of other procedures.

The position of the rib with respect to instantiated neighbors is used to adjust horizontal and vertical scale factors governing the predicted size of the ribcage. The position of the rib affects the predicted range of locations for other unfound ribs. The shape of the rib also affects the search region for uninstantiated rib neighbors.

If some evidence is found for the rib, but not enough to warrant an instantiation, the rib hypothesis is left on the active list and the rib model node is not instantiated. Rib hypotheses left on the active list will be reconsidered by the executive, which may try them again on the basis of new evidence.

The sequence of figures (Fig. A2.1, p. 503) shows a few steps in the finding of ribs using this program. Figure A2.1a shows the input data. A2.1b shows rectangles enclosing the lung field and the initial area to be searched for a particular rib which is usually findable. Only one rib-finding procedure is applicable for ribs with no neighbors found, so it is invoked and the rib shown by dark boxes in Fig. A2.1b is found. Predicted locations for neighboring ribs are generated and are used in order by the executive which invokes the rib-finding procedures in order of expected utility (A2.1c-e). Predicted locations are shown by dots, actual locations by crosses; in Fig. A2.1f, all modelled ribs are found. The type of procedure that found the rib is denoted by the symbol used to draw in the rib. Figure A2.1f shows the final rib borders superimposed on the data.

A2.3.3 Pattern Directed Invocation

Considerable attention has been focused recently on pattern directed systems (see, e.g., [Waterman and Hayes-Roth 1978]). Another common example of a pattern directed system is the production system, discussed in Section 12.3. The idea behind a pattern directed system is that a procedure will be activated not when its

name is invoked, but when a key situation occurs. These systems have in common that their activity is guided by the appearance of "patterns" of data in either input or memory. Broadly construed, all data forms patterns, and hence patterns guide any computation. This section is concerned with a definition of patterns as something very much smaller than the entire data set, together with the specification of control mechanisms that make use of them.

Pattern directed systems have three components.

1. A data structure or data base containing modifiable items whose structure may be defined in terms of patterns
2. Pattern-directed modules that match patterns in the data structure
3. A controlling executive that selects modules that match patterns and activates them

A popular name for a pattern-directed procedure is a *demon.* Demons were named originally by Selfridge [Selfridge 1959]. They are used successfully in many AI programs, notably in a natural language understanding system [Charniak 1972]. Generally, a demon is a program which is associated with a *pattern* that describes part of the knowledge base (usually the pattern is closely related to the form of "items" in a data base). When a part of the knowledge base matching the pattern is added, modified, or deleted, the demon runs "automatically." It is as if the demon were constantly watching the data base waiting for information associated with certain patterns to change. Of course, in most implementations on conventional computers, demons are not always actively watching. Equivalent behavior is simulated by having the demons register their interests with the system routines that access the data base. Then upon access, the system can check for demon activation conditions and arrange for the interested demons to be run when the data base changes.

Advanced languages that support a sophisticated data base often provide demon facilities, which are variously known as if-added and if-removed procedures, antecedent theorems, traps, or triggers.

A2.3.4 Blackboard Systems

In artificial intelligence literature, a "blackboard" is a special kind of globally accessible data base. The term first became prominent in the context of a large pattern directed system to understand human speech [Erman and Lesser 1975; Erman et al. 1980]. More recently, blackboards have been used as a vision control system [Hanson and Riseman 1978]. Blackboards often have mechanisms associated with them for invoking demons and synchronizing their activities. One can appreciate that programming with demons can be difficult. Since general patterns are being used, one can never be sure exactly when a pattern directed procedure will be activated; often they can be activated in incorrect or bizarre sequences not anticipated by their designer. Blackboards attempt to alleviate this uncertainly by controlling the matching process in two ways:

1. Blackboards represent the current part of the model that is being associated with image data;

2. Blackboards incorporate rules that determine which specialized subsystems of demons are likely to be needed for the current job. This structuring of the data base of procedures increases efficiency and loosely corresponds to a "mental set."

These two ideas are illustrated by Figs. A2.2 and A2.3 [Hanson and Riseman 1978]. Figure A2.2 shows the concept of a blackboard as a repository for only model-image bindings. Figure A2.3 shows transformations between model entities that are used to select appropriate groups of demons.

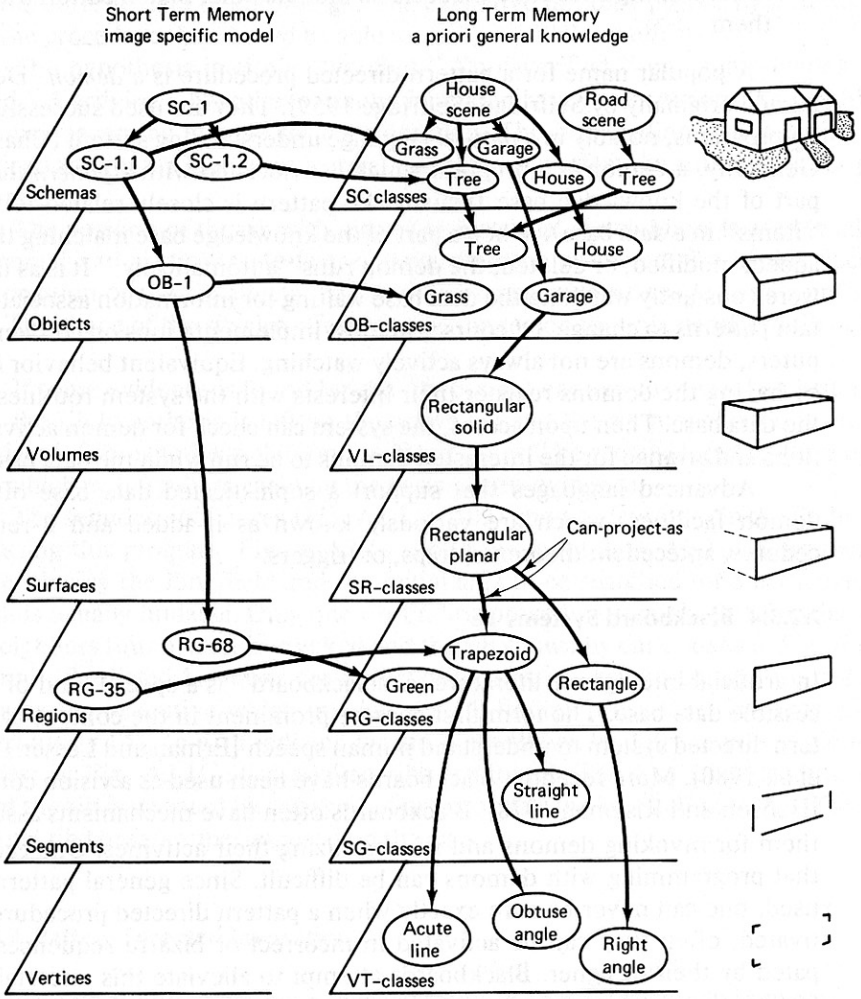

Fig. A2.2 An implementation of the blackboard concept. Here the blackboard is called Short Term Memory; it holds a partial interpretation of a specific image.

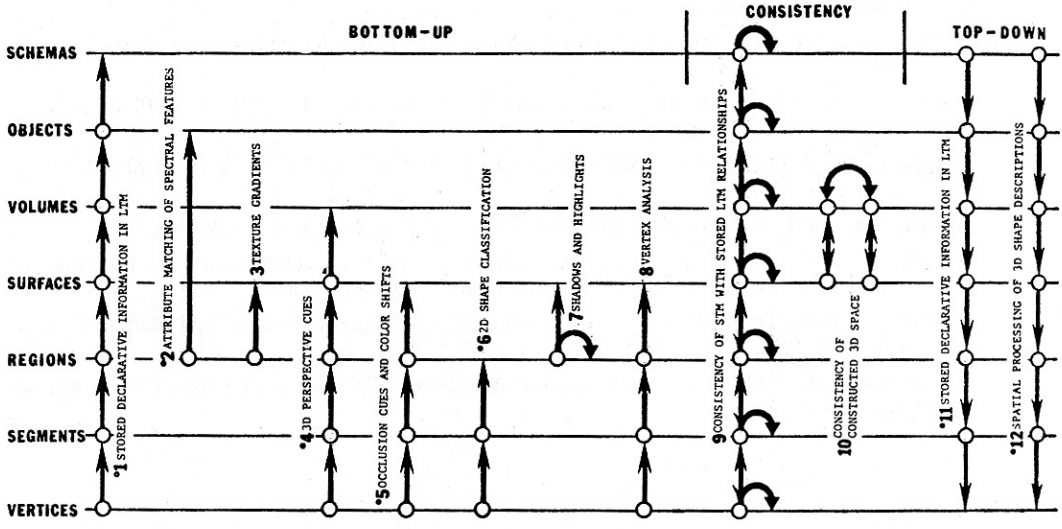

Fig. A2.3 Paths for hypothesis flow, showing transformations between model entities and the sorts of knowledge needed for the transformations.

REFERENCES

AHO, A. V., J. E. HOPCROFT and J. D. ULLMAN. *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1974.

ASTRAHAN, M. M. et al. "System R: A relational approach to data base management." IBM Research Lab, February 1976.

BALLARD, D. H. "Model-directed detection of ribs in chest radiographs." Proceedings, Fourth IJCPR, Kyoto, Japan, 1978.

BIRTWHISTLE, G. et al. *Simula Begin.* Philadelphia: Auerbach, 1973.

BOBROW, D. G. and B. RAPHAEL. "New programming languages for artificial intelligence." *Computing Surveys 6*, 3, September 1974, 155–174.

CHARNIAK, E. "Towards a model of children's story comprehension." AI-TR-266, AI Lab, MIT, 1972.

ERMAN, L. D. and V. R. LESSER. "A multi-level organization for problem solving using many diverse cooperating sources of knowledge." *Proc.*, 4th IJCAI, September 1975, 483–490.

ERMAN, L. D., F. HAYES-ROTH, V. R. LESSER, and D. R. REDDY. "The HEARSAY-II speech-understanding system: Integrating knowledge to resolve uncertainty." *Computing Surveys 12*, 2, June 1980, 213–253.

FAHLMAN, S. E. "A planning system for robot construction tasks." *Artificial Intelligence 5*, 1, Spring 1974, 1–49.

FELDMAN, J. A. "High-level programming for distributed computing." *Comm. ACM 22*, 6, July 1979, 363–368.

FELDMAN, J. A. and R. F. SPROULL. "System support for the Stanford hand-eye system." *Proc.*, 2nd IJCAI, September 1971, 183–189.

FLOYD, R. W. "The paradigms of programming." *Comm. ACM 22*, 8, August 1979, 455–460.

GOLDBERG, A. and A. KAY (Eds). "SMALLTALK-72 Instruction Manual." SSL 76-6, Xerox PARC, Palo Alto, CA, 1976.

HANSON, A. R. and E. M. RISEMAN. "Visions: A computer system for interpreting scenes." In *CVS*, 1978.

HEWITT, C. "Description and theoretical analysis (using schemata) of PLANNER" (Ph.D. dissertation). AI-TR-258, AI Lab, MIT, 1972.

HEWITT, C. and B. SMITH. "Towards a programming apprentice." *IEEE Trans. Software Engineering. 1*, 1, March 1975, 26–45.

KNUTH, D. E. *The Art of Computer Programming*, Vol. 1. Reading, MA: Addison-Wesley, 1973.

SELFRIDGE, O. "Pandemonium, a paradigm for learning." In *Proc.*, Symp. on the Mechanisation of Thought Processes, National Physical Laboratory, Teddington, England, 1959.

SUSSMAN, G. J. and D. MCDERMOTT. "Why conniving is better than planning." AI Memo 255A, AI Lab, MIT, 1972.

WATERMAN, D. A. and F. HAYES-ROTH (Eds.). *Pattern-Directed Inference Systems.* New York: Academic Press, 1978.