Chapter 3

Concepts of Artificial Intelligence

3.1 Introduction

Artificial Intelligence (AI) is an area of research that is concerned with designing and programming machines to realise tasks that are usually achieved by intelligent organisms. More specifically it attempts to model the cognitive process of human beings in solving problems. According to Chandrasekaran [Chan90], intelligence is a process of manipulating representations of the world and ideas. In this respect, workers in AI have produced a variety of tools and techniques to assist in domain oriented applications.

The field of AI can be subdivided into three broad areas: natural language processing, robotics, and knowledge-based systems [Owen87]. The last two areas are of interest for behavioural animation and therefore this chapter will concentrate on relevant topics in these areas. Firstly the terminology used in AI will be considered, followed by discussions on the nature of actions performed by intelligent beings. Then we present some principles of problem solving methods employed in knowledge-based systems. It also includes some issues relating to data representation and control.

3.2 AI Nomenclature

In order to present a common language for the readers of this thesis and to explain the approach of the current framework in future chapters, some AI concepts and methods will be reviewed and whenever possible exemplified.

It is important to observe that there are some differences among AI practitioners in explaining terms and concepts that have been used for many years in AI. Authors of

several AI books and papers [Wilk88, Camp86, Part86] do not hide their frustration at this problem. Not uncommonly concepts are introduced to the readers through the use of examples before addressing to the main topic. Others simply assume in advance that the reader is familiar with the nomenclature used. Evidence of the trouble in defining or formalising some of the AI terms is given by Wilkins [Wilk88]:

> "Reasoning about actions is a necessary element of intelligent behaviour. A person can scarcely participate in a conversation or go to the store for groceries without reasoning about how actions taken will affect the surrounding world... Decades of research in Artificial Intelligence (AI) and related disciplines have shown this particular human capability to be extremely difficult to formalize..."

Perhaps this difficulty stems from the following realisation by Campbell [Camp86]:

> "Many of the uncertainties about the exact nature of AI are consequences of the fact that it is a science which does not quite fit in with other categories of sciences... One of the features which distinguishes AI from most other sciences is that it refers to objects (programs or conceptual structures capable of being realized in programs) which are created by humans rather than objects having a prior natural existence... The psychological or cognitive-modelling side of the subject is a popular area for debate among AI workers, but the debate generates more heat than light. Seen from the viewpoint of most other sciences, the experimental testability of the computational models' predictions or behaviour against human behaviour is extremely doubtful except in a limited area (e.g. parts of the study of vision) where at least as much neurophysiology as psychology seems to be involved. This is a controversial opinion!"

Finally, difficulties with terminology is not an exclusive problem of AI, Bergeron [Berg83] has acknowledged the same problem in computer animation in the early stage:

> "...Animation is probably the most 'esoteric' field of computer graphics. From place to place, computer animators do not speak the same language..."

3.3 Reasoning about Actions

One of the most evident characteristics of the real world is that it is inhabited by organisms with the capability to perform *actions*. These organisms, which can be animals, robots, etc., are called *agents* if they can carry out activities. For our purposes, the agent in question is a robot with humanoid appearance and implemented as a program. In contrast, there is a sub-field in AI that uses the concept of *actor* in a close

context of the *agent* [Hewi73]. The denomination of *actor* has been used since the early stage of computer animation [Reyn82] and currently is used synonymously with *agent*. Both are equivalent terms in the sense that they perform actions.

As Georgeff [Geor90] explains that in the course of actions, at any given moment, the world is in one of a potentially infinite number of *states* or *situations*. A *world state* may be viewed as a snapshot of the world at a given instant of time. And because a state is a description of the world for an instant of time, there is potentially a collection of facts that can be observed from a state. More specifically, amidst all these infinite states, the *initial* and the *final states* of an action are the relevant ones for that action. It can be understood that the *initial state* gives some of the condition for an action to be initiated and the *final state* contains the outcome at the conclusion of the action. The final state is also called the *goal state* of the action, or simply the *goal*, because that state of the world will include the results achieved by that action.

According to Georgeff, an *event* is any behaviour that occurs in anytime, while an *action* is a special case of an event in the sense that it is caused by the agent itself in an intentional way. From the agent point of view, any other action that is not caused by that agent or that is caused unintentionally in the nature is called *event*. For example, *John standing up* within the environment is an event for Mary. Thus, Mary may decide to sit down as a consequence of the event *John stands up*. In nature, a tree shedding its leaves is an event but not an action. Despite this differentiation, events and actions can be used interchangeably as synonymous.

The simplest actions are accomplished by a single *operator*, for example the *STRIPS operator*[1]. An example of a typical STRIPS operator is shown in Figure 3-1 [Tate90]. The *pickup* operation has a parameter *x* on which it will be executed if the precondition holds. The precondition is a logical formula specified by a conjunction of facts, or relationships, that must be satisfied to allow the lifting of an object *x*. If the precondition is satisfied the operator is said to be *applicable* to that world description, thus the object

[1] STRIPS is one the first problem-solving systems which introduced the state-based representation used in planning for big number of system.

x is lifted and, as a result of the operation, some facts are deleted and new facts are added to the world state at the conclusion of the operation. The accomplishment, or goal, of an operator is the change thus obtained in the world state (e.g., HOLDING(x)). Such an operation is a deterministic one, since the operator is not interrupted as it executes.

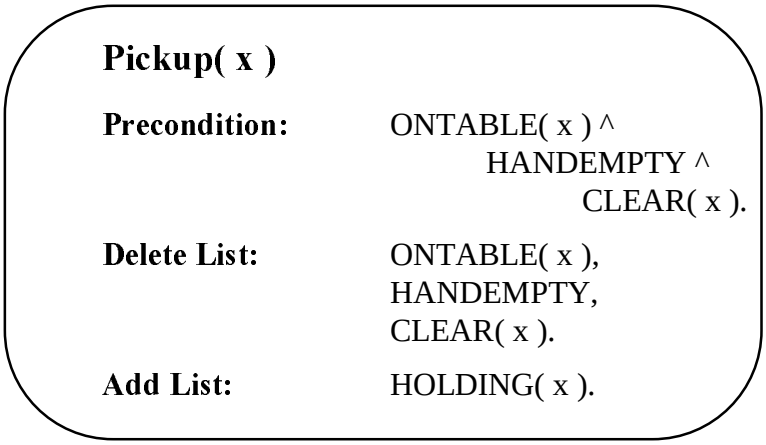

Figure 3-1: A typical STRIPS operator.

From the example in Figure 3-1, it is understood that an action stretches for a period of time where the initial state, the precondition of the action, remains the same from the beginning of the action until near its conclusion. When the action ends, a list of facts are deleted and replaced by another; and facts resulting from this update may take part of the initial state of subsequent actions. Such an assumption about discrete changes of states is known as *the STRIPS assumption*. In a dynamic world where events are typically continuous this assumption may not work perfectly because of the mutual interference that might occur during the performance of an action and the assumptions about the world may no longer be valid. Georgeff acknowledges that no one has yet provided adequate semantics for this problem. However, this assumption works well if we restrict to domains with agents performing non-concurrent activities.

A simple example in which the change of states may be observed is in shown in Figure 3-2. Suppose that John is in the *seated* state and he is ordered to perform a *stand up* action. At the end of the action he will be in the *standing up* state. Additionally, it can be observed that both the action "to stand up" and the goal state "to be in the standing up position" can be used interchangeably. In this sense both, the pair of states (initial

and goal) and the action, are dual to each other in that one does not exist without the other. That is, there are occasions that it seems to be easier to ask what to do rather than to ask what we want the world states to be, and vice-versa.

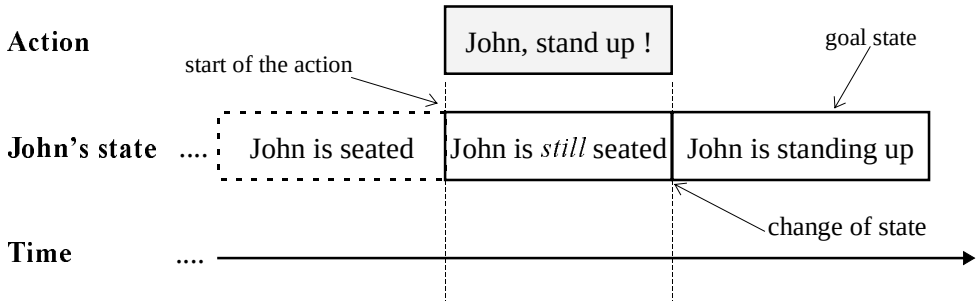

Figure 3-2: Changes in John's world state as he accomplishes an action.

More common actions are accomplished by a sequence of operations, specially when the goals pursued are not so evident as in the single operator case. These operations form a *plan* of action which give an agent a more complex behaviour. A plan is said to be a solution to a given problem if it is applicable in the problem's initial state and, if after execution of the plan, the goal is true [Tate90]. The applicability of a plan as a whole is equivalent to that of a single operator with the exception of the number of operators. That is, if all the conditions of the first operator in a plan hold then the plan is said to be applicable. Figure 3-3 compares the applicability of both a single operator action and a multiple operators action for a given world state. The successive application of the operators produce intermediate state descriptions, until eventually the last operator yields the states specified as goals.

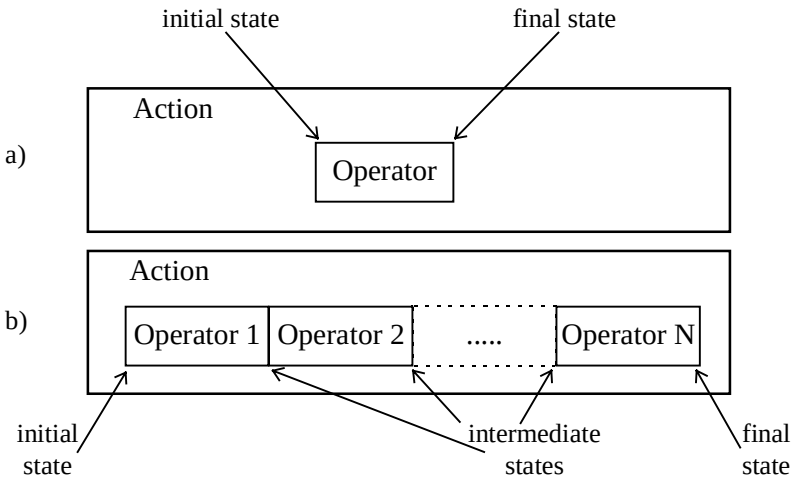

Figure 3-3: Applicability of single and multiple operators cases.

3.4 Plan Synthesis

Plan synthesis concerns the construction of a course of action for an agent to achieve some specified goals [Geor90]. This process is commonly known as *planning*. The system that has the capability to construct plans is called a planning system or planner. It is also called *problem solver* because its activity is to find sequences of actions that solve problems, that is, goal satisfaction.

There is a diversity of problem solving systems. Generally what differentiates one planner from another is the reasoning method (search mechanism) that operates in the construction of plans. Cohen et al. [Cohe83] identify four main approaches to planning which are found in most of AI problem-solving system: *non-hierarchical planning*, *hierarchical planning*, *script-based planning*, and *opportunistic planning*. Most of the planning approaches are typically hierarchical with the exception of non-hierarchical planning. Hierarchy in this context implies a relationship of dependence between one element with a few other sub-elements.

3.4.1 Non-hierarchical Planning

Non-hierarchical planning is the most primitive approach in which the planner only deals with a simple form of actions (the operators) without resorting to any form of abstraction as in the hierarchical case. Given a set of one or more goals, the planner chooses the actions with goals that match with those sought. These actions form the initial plan from which a consistent plan will be developed by the planner. This is attempted by re-ordering the actions or interleaving new actions into the working plan if there are gaps between goals.

Figure 3-4 shows a simple example of planning in the blocks world [Wilk88]. For example, to achieve the result of a block A on top of a block B a robot must put block A on block B. The condition to accomplish this action is that the top of both blocks are clear. In the case where the condition is not completely satisfied (a block C is on top of a block B), it is necessary to create a new goal to satisfy the condition of having the top of B cleared. This kind of planning blindly attempts a number of alternatives and, if it

succeeds, it may create an excessively long sequence of operations to achieve a simple goal.

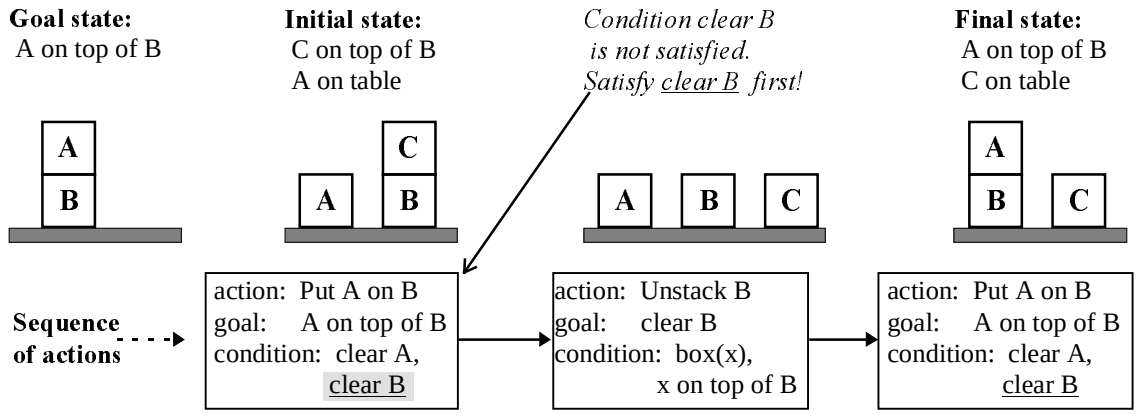

Figure 3-4: Example of the non-hierarchical planning approach.

3.4.2 Hierarchical Planning

Hierarchical planning overcomes the problem of lack of co-ordination in the non-hierarchical case through the introduction of abstraction into the action plan. The plan for an action is composed of a sequence of sub-actions rather than solely of operators. Therefore an abstract, or complex, action can be represented by a plan composed of a few simpler actions rather than a long sequence of operators. Moreover, by doing so, the building of the plan evolves in several stages but in a controlled way until a complete plan is developed. At the first stage a sketch of a general but vague plan, composed of a few abstract actions, is created. In the second stage, the vague parts of the plan are further replaced by more specific actions. These changes are thus successively incorporated into the evolving plan until a complete sequence of operators is achieved. Figure 3-5 shows the development of a high-level plan of one action down to a full sequence of operators. The shaded nodes form the partial plans of each stage and they are replaced by blank nodes as they are expanded into new nodes. The new nodes are numbered according to the stage at which they are created. However, the choice of a node to expand depends on the established criteria. If at any moment the expansion process fails it has to backtrack, that is, undo the latest expansion, and attempt another alternative. One way to avoid excessive backtracking is by imposing constraints in the earlier stages of plan expansion rather than testing the operator's preconditions.

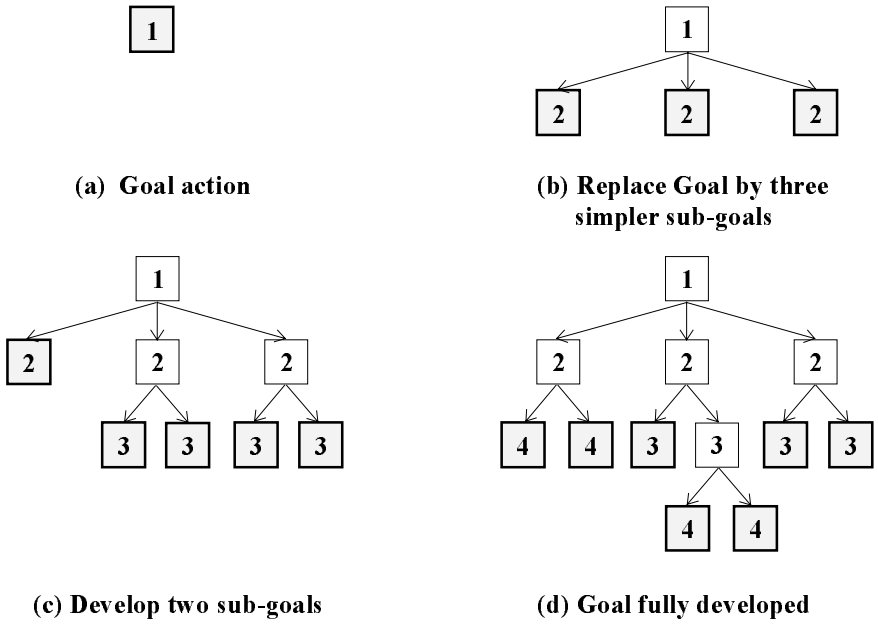

Figure 3-5: Evolution of a plan in an hierarchical model.

3.4.3 Script-based Planning

Script-based planning approach is similar to hierarchical planning. Plans are expanded in successive levels but instead of working out a plan for a vague node, the planner selects a suitable plan from a collection of predefined plans for that goal. That is, an action is stereotyped by a limited number of pre-defined plans, each plan is prescribed to deal with one typical situation. The choice for a plan is made as the constraints attached to it are satisfied. Each step of the plan is then further refined into a more detailed frame. This kind of plan is called a *skeletal plan* because it formulates a general strategy of how an action is to be solved. Because the number of alternative plans is limited the planning process is fast. This kind of approach is suitable for stereotyped situations or ones which are difficult to deduce by automated reasoning, therefore, a solution must be explicitly specified within the plan and this task is done by a human expert.

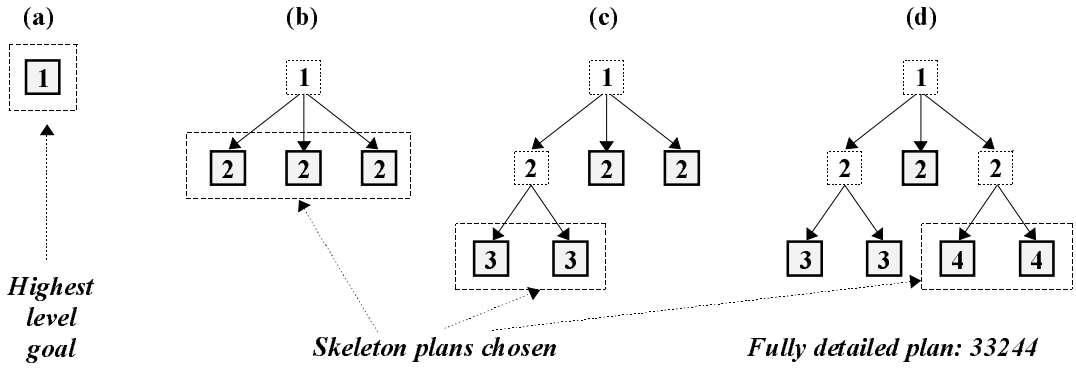

Figure 3-6: Example of Script-based planning.

3.4.4 Opportunistic Planning

Opportunistic planning is characterised as a bottom-up process in contrast to the ones discussed above. However, hierarchy is preserved. As the data and facts about events are put in the system memory, patterns of these data are recognised as new facts or goals, as opportunities arise. In this process new facts are included in the memory and others are removed. This clusters, or reduces, existing information into a smaller number of facts (an analogy to the "agglutination" of smaller islands into bigger ones can be made). Eventually no more pieces of data are left in the system memory or no pattern is recognised. Figure 3-7 presents a sequence of goal reductions that are performed until the sought goal is achieved. Starting from a configuration given in the frame (a), frame (e) can be obtained as a process of reduction (or pattern recognition) by applying the operators provided on the left of Figure 3-7.

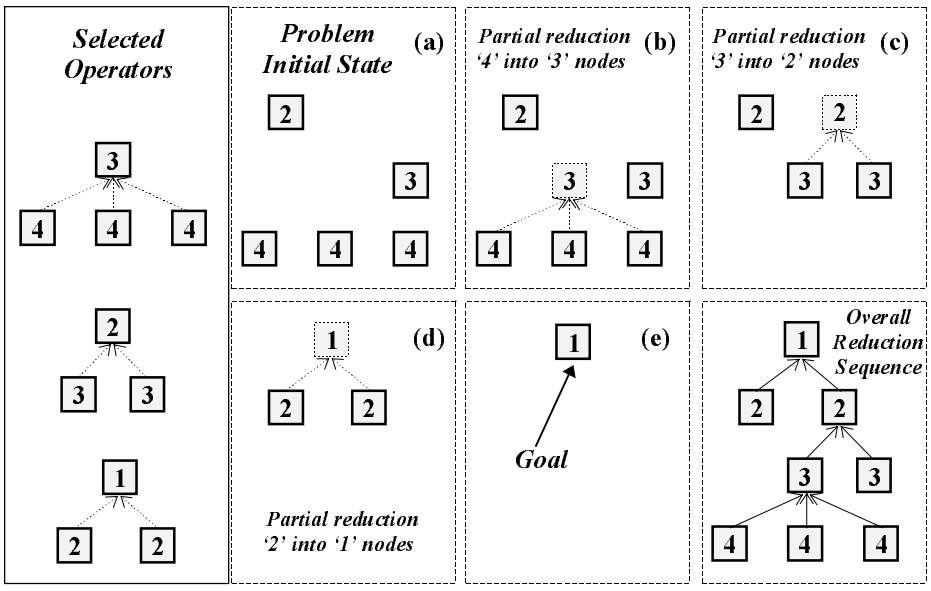

Figure 3-7: Example of opportunistic planning.

3.5 Knowledge-Based Systems

Knowledge-based systems (KBSs) are computer programs that implement the cognitive model of a human expert to solve problems in specific domains of application. Usually they are designed to help people with tasks involving uncertainty and imprecision, and which require judgement and knowledge [Hart92]. In the case of an application in animation, the system comprises the computational model of the animated objects, the knowledge about the behaviour of the objects, and the coordination of the activities of the animation environment. In KBS the knowledge and the concepts about a domain are conveniently represented as symbols which are readable by humans and suitable for manipulation by a program.

The problem with typical KBSs applications is that the knowledge about their domain is well known but the description or representation may not be straightforward. Such knowledge is in a constant process of refinement, updating, or re-organisation; as experience is gained. Nevertheless, the way the knowledge is used usually remains the same. A KBS is thus characterised by two main components: the *application knowledge-base* and the *control framework* [Brow89]. The application knowledge-base is typically *declarative* in the sense that knowledge is stated, or explicitly represented. Whereas the control framework is comprised of application-oriented procedures that know how to manipulate that knowledge. These issues are reviewed in the next sections. Furthermore, a KBS commonly runs under a shell environment that provides features and functionalities of a working environment such as the inference capability and the workspace for data storage (Figure 3-8).

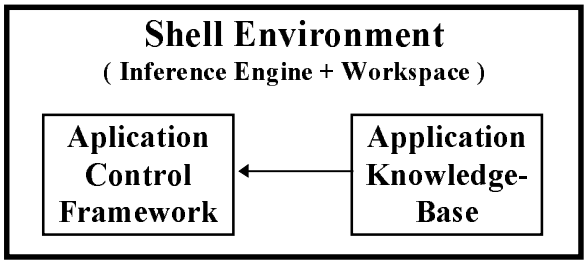

Figure 3-8: Typical components of a knowledge-based system.

3.6 Knowledge Representation

The aim of knowledge representation is to write down descriptions of the world in such a way that the system's inference mechanism can come to new conclusions about the application domain by manipulating these descriptions [Fike85, Ring88]. It is equally important that the human operator can readily enter new information and interpret the conclusions. One requirement is the choice of the language that permits knowledge to be expressed symbolically. Languages like Lisp and Prolog are suitable for symbolic manipulation. The other requirement is to organise the knowledge in a structure that permits the problem solver to process it effectively.

In AI, the knowledge about the problem it is solving is basically represented in terms of predicates, frames, rules, and procedures.

3.6.1 Literals

Literals are symbolic names that identify or represent objects, concepts, or meanings. The literal is the basic element which are nodes in the composition of other knowledge representation structures such as predicates, rules, frames, semantic networks, etc. Examples of literals are *glass, counter, location_of, front*, *person, is_a*, etc.

3.6.2 Variables

Variables can store simple data such as literals or complex data structures such as lists and predicates. A variable name is denoted by a word in which at least the first letter is capitalised. Variables accept instantiation and consequently they may connect predicates with common variables. For example, in the predicates, *condition_of(Glass, clean)* and *isa_glass(Glass), Glass* could be substituted by any literal that satisfies both predicates, for instance, *tall_glass* or *my_glass*.

3.6.3 Predicates

Predicates, or facts, are synonymous with the relationships that associate one or more literals in the representation of knowledge. These relationships connect symbolic objects, or concepts, and associate them with semantic information. Such relationships can be represented in different forms denoting the same fact. Therefore, a naming

discipline has to be adopted and the arity and ordering must be consistent throughout the application domain. The manifestation of a relationship occurs once it has been inserted into the working memory, so that future operations can consider it for inference. In the light of subsequent events, facts may be deleted. Some examples of sentences with the corresponding predicates are given:

given sentence: *glass is an object*
in predicate form: *object(glass)* or *isa(glass, object)*

given sentence: *the glass is in the clean condition*
in predicate form: *glass(condition, clean)* or *condition_of(glass, clean)*

given sentence: *the my_glass is on the counter on the left*
in predicate form: *location_of(my_glass, counter1, left)*

3.6.4 Semantic Networks

A semantic network is a net or graph of nodes joined by links. The nodes of the net are literals and the links are labelled with semantic information representing relations. This representation not only captures definitions of concepts but also provides access to other concepts across the net [Rand88]. Facts are examples of simple nets with one item of semantic information and the connection of a diversity of facts builds up a semantic network. Much research has been done in semantic networks because of its potential complexity, however, despite many developments, Randal [Rand88] pointed out that semantic networks are not sufficient for knowledge representation as new "extensions" to its notation are constantly being added. A simple example is shown in Figure 3-9.

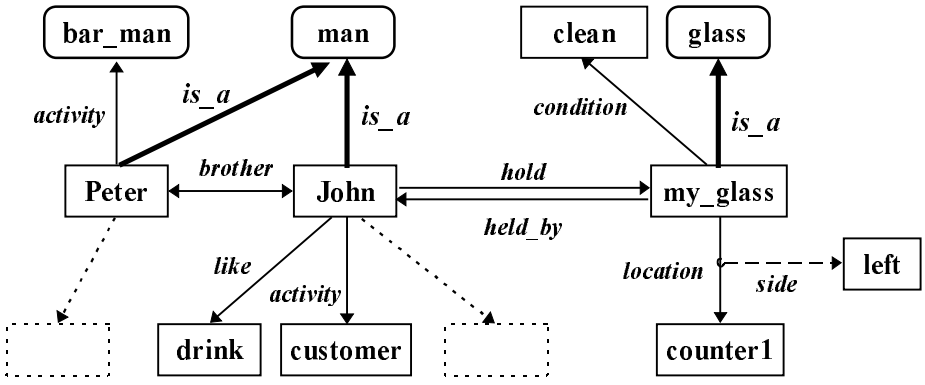

Figure 3-9: Fragment of a semantic network.

3.6.5 Frames

Frames, also known as schemata, are knowledge representations widely used in AI and robotics. They are a very useful form of stereotyping knowledge about objects, or concepts, in terms of attributes that are inherent. The frame groups these attributes which are individually described in terms of the *slot* and the *filler*. A *slot* is a literal that identifies one feature, or attribute, of the modelled concept and the *filler* is the value assigned to the slot. The knowledge stored in a frame constitutes a variable number of slots filled with default values. New attributes are added to the frame as the application evolves. A frame structure is thus very similar to the *struct* data type in the C language or the *record* in the Pascal language. The value held by a slot can also be a frame name or an object name. In these cases, when a frame has links to other frames, a *structural link* is established. Both the structural link and the relational link have considerable importance in the reasoning process because the chaining of these links gives access to new information.

The *default values* are chosen as typical values a frame might assume. In Figure 3-10, for example, the *colour* slot of the frame *entity* has the default value *blue*. An example of a typical situation occurs in the usage of the *place* slot of an action. The *place* attributes may have different uses depending on the situation it is used. For example, *counter* and *table* are places that a person can go to have a drink if he is a customer, or to serve if he is a barman. During the reasoning process if the retrieval of an item of information from a slot occurs before its updating, that is, if no current values had previously been set, then the default values will be retrieved instead.

```
frame entity;
  default name is nobody and
  default colour is blue and
  default location is pos(0,0,0) and
  default direction is 0 and
  default place is world .

frame glass is an entity;
  default type is glass and
  default place is counter and
  default condition is clean and
...

frame counter is an object, surface ;
  default type is counter and
  default state is available and
  default colour is blue and
...
```

```
frame person is an entity;
  default type is person and
  default posture is stand_up and
  default holding is nothing and
....

frame barman is a person;
  default activity is barman and
  default counter is counter and
...

frame waiter is a person;
  default activity is waiter and
  default counter is counter and
...

frame object is an entity;
  default user is nobody and
...
```

Figure 3-10: Frame.

Inheritance

Frames are an important mechanism for inheritance that permit the organisation of classes and sub-classes of objects and concepts in a similar way to the object-oriented approach. That is, the hierarchical organisation of classes of objects can be reflected in the frames by establishing parent/children relationships (links) as shown in the Figure 3-10. Child frames are derived from parents frames and, in this process, the slots with default values of the parent frames are inherited. Furthermore, the addition of slots in the child frames turn them into a specialisation of the parent frame. For example, a generic frame, such as the *entity* frame in the Figure 3-10, has some attributes (or slots) which are common to a group of frames, such as the *person* frame and the *object* frame. These frames structure classes into a hierarchy such as that shown in Figure 3-11. In another example, both frames *barman* and *waiter* are specialisations of the *person* frame, differing in the default *activity* and in the addition of some slots that characterise their activities.

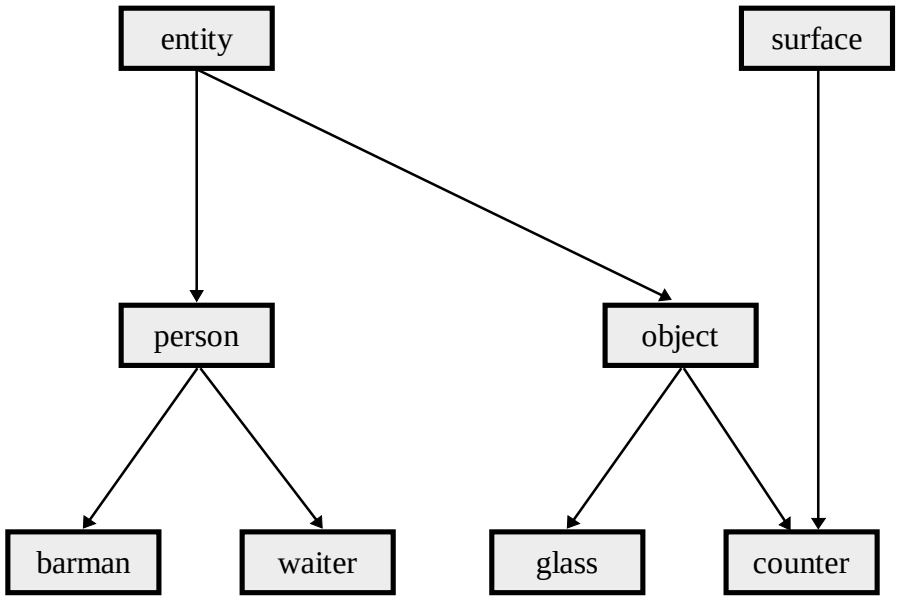

Figure 3-11: Frame inheritance: arrows point to new inherited classes.

Instances

An instance of a frame is an object that is an exact copy of the frame and has the functionalities conceived for the original frame. In the absence of current values, the default values are fetched from the original frame. There is no difference between frame and instance with regard to default values, but the creation of instances for each new object becomes necessary because: distinct objects require the allocation of distinct storage for holding distinct current values; the original frame is preserved when new objects of the same class are needed; frames stereotype classes of objects and cannot change. For example, two characters are created in an animation scenario, *John* and *Mary*. Both can be instances of the *waiter* frame, *John* might be *holding* a glass while *Mary* is empty-handed.

3.6.6 Production Rules

Production rules, or rules, are a form of knowledge representation that associate one collection of facts with another. If the facts of the first part of a rule are true then the action in the second part is made to occur. The format of a rule varies from one implementation to another. For example, the STRIPS operator is a kind of rule that is fired when its preconditions are satisfied. The structure of a rule is typically represented as:

RULE R_k

IF A_1, A_2, ..., A_M THEN C_1, C_2, ..., C_N

where the first part of the rule, A_i (i = 1, 2, ..., M) is the *condition* part; and, C_j (j = 1, 2, ..., N) are the *actions* part of the rule named R_k. The condition part, also known as the *antecedent,* is the requirement for the selection of the rule. The condition part can also be regarded as a pattern to be matched with items from the system database. The action part, known as the *consequent,* is the change to be applied to the environment database.

One of the main applications of rule representation is to find answers to a problem (goal). The other example is the use of a rule as a data operator, that is, patterns of data in the environment database are recognised by rules and transformed. The transformation can be data composition, decomposition, or modification, etc. In such cases, the process stops when no pattern can be recognised.

3.6.7 Procedures

Procedures as a form of knowledge representation are typically written in declarative programming languages such as Lisp and Prolog. Procedures are small programs that know how to do specific things, how to proceed in well-specified situations [Barr81]. More specifically, a procedure in Prolog is a conventional rule that works backwards, that is, the consequent is true if the antecedents can be verified (Figure 3-12). Such a kind of rule is a Prolog procedure called a *clause.* The head (goal) of a clause is equivalent to the consequent part of a rule and the body is equivalent to the antecedents. Therefore, the body of a Prolog clause is a sequence of facts that chains a series of relationships and the information thus obtained is sent out. The head is a fact that acts as an interface that has a variable number of parameters that either flow in or out.

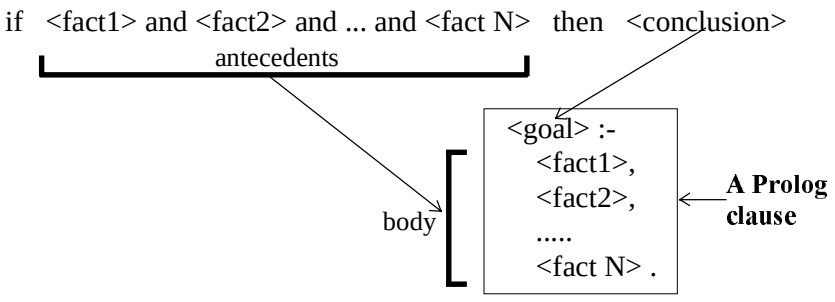

Figure 3-12: Prolog clause as rule.

3.7 The Reasoning Process

The reasoning process is carried out by an inference engine. The inference engine, in its turn, is the active program in a knowledge-based system that interprets the queries, or commands, input by the user. For each (non-trivial) query entered, the inference engine starts a series of rule chaining operations that eventually come up with an answer. That is, the inference engine is provided with an automatic mechanism that is triggered in response to every individual operation being considered throughout the inference process.

Rules are organised in bundles called *rulesets* and each ruleset has an associated control strategy. The main components in overall rule control are *rule chaining* and *conflict resolution*.

3.7.1 Rule Chaining

The control of rule chaining determines how the process of reaching solutions is achieved. The two main types of chaining control in the reasoning process are *forward-chaining* and *backward-chaining*.

Forward-chaining is also referred to as the *data-driven* or *bottom-up* process and it is typically used in expert systems. Rules are triggered as the patterns formed by facts in the condition parts are matched in the context. The execution of rules modifies the context database by adding new facts and removing old ones. Thus, every newly created context can in turn be used by rules to derive new facts. The forward process chains a succession of rules and stops when either the goal is achieved or no more rules can be triggered. As Figure 3-13 shows, the inference engine triggers rules systematically until the goal is eventually found as part of one of the context database.

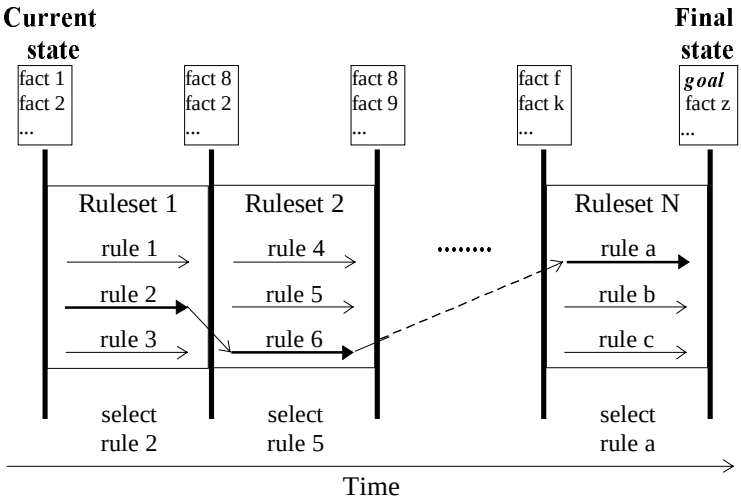

Figure 3-13: Example of forward chaining process.

Backward-chaining is also referred to as the *goal-driven* or *top-down* process. In the trivial case, if the goal to be achieved is found in the context database then the goal is successful and no further chaining is required. Otherwise, it attempts to find a rule whose consequents contains the goal. If such a rule is found then the antecedents of the rule will be the "new goals" (or sub-goals) to be achieved and the process continues until all antecedents can be verified in the environment. The purpose of the backward-chaining is to support or refute a goal by searching facts in the database. Figure 3-14 exemplifies the backward chaining from a goal. In this case it takes two steps to infer that G is supported by the context.

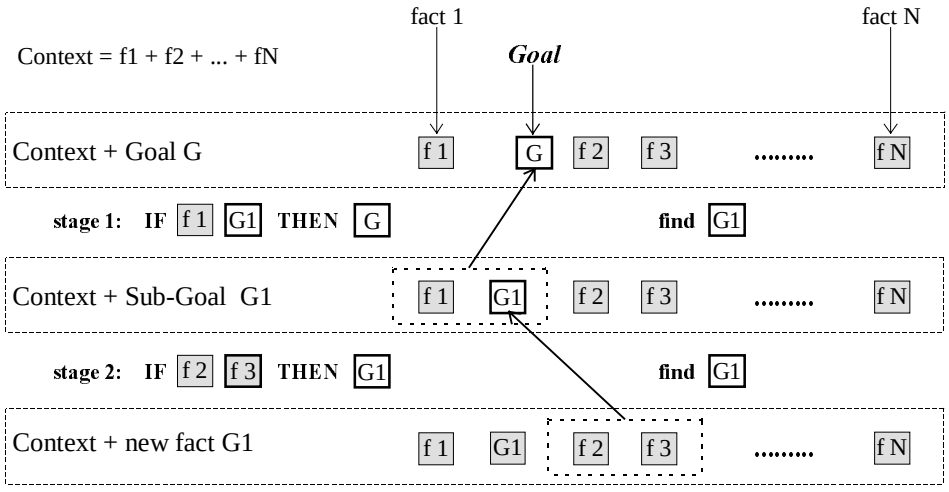

Figure 3-14: Example of backward chaining from a goal.

3.7.2 Conflict Resolution

Conflict resolution is required in situations where more than one rule has its condition satisfied, in this case a rule selection scheme must be used. The selection scheme controls the order in which the rules are considered. Examples of these schemes are *first come first served*, *conflict resolution scoring system*, *conflict resolution with a threshold value*, etc. The first-come-first-served scheme simply considers rules in a fixed order until finding the first rule whose condition is satisfied. The conflict resolution by scoring system assesses the scores of the rules whose conditions are satisfied and triggers the highest scored rule. The conflict resolution with a threshold value is similar to the previous one except that it triggers the first rule whose score is greater than the threshold value. Both conflict resolution schemes depend very much on the particular heuristics employed to score the rules.

As a result of the conflict resolution scheme, the search in the reasoning process is conducted in two directions: *depth first* and *breadth first* (Figure 3-15). In the *depth first* search order, the reasoning process branches to the first node of the level and proceeds in this fashion until the goal is reached or a terminating condition is satisfied. The traversal proceeds by backtracking up to the next node of the previous level and the depth first search is applied to this next node. In the case of *breadth first* search order the inference engine sweeps all the applicable rules in that level of the hierarchy before pursuing the sub-goals of the following level. Figure 3-15 exemplifies the branching (search) in a backward-chaining process but it can also be applied to the forward-chaining process.

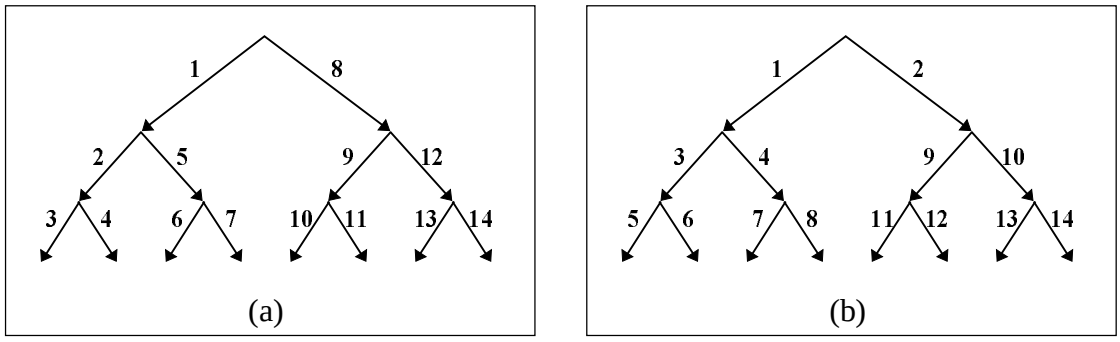

Figure 3-15: Backward-chaining: (a) depth first order; (b) breadth first order.

3.8 Frameworks in KBS

In this section two typical AI approaches for KBS are summarised, namely, expert systems and blackboard systems. Expert systems (ESs) are briefly presented as they are essentially built as a collection from the AI paradigms reviewed in this chapter. The blackboard model (BBM) is a new approach that complements the ESs in areas of applications which are not sufficiently assisted by the ES model. In fact, the BBM is a conceptual model that subsumes the ES and has additional features that permit it to deal with problems more complex than those covered by ESs. The BBM is probably the most general model for building knowledge system architecture because it accepts nearly all engineering tools (ES, chaining control) [Enge88]. A special attention is given to the BBM as this model has inspired the proposed animation framework.

3.8.1 Expert Systems

ESs are synonymous to knowledge-based systems as they have been the most widely known programs in AI for problem solving [Tzaf90]. Thus, the AI paradigms reviewed in this chapter are also part of the ESs in general. According to Barr et al. [Barr89], the two broad classes of problems addressed by ESs in general are:

- Problems of interpreting data to analyse a situation
- Problems of constructing a solution within specified constraints.

Barr et al. have also pointed out three reasons for using the methods of expert systems:

- complexity - in the initial stage of the development of the system the description of the problem is oversimplified and its feasibility becomes known as the first prototypes are built to solve a small scale version of the problem. In a second stage, the expert system evolves incrementally with the better understanding of the problem.
- interpretation - any data or program can be computed (evaluated) by the running system or can be queried by users to examine information during run-time session.
- knowledge - the ability to specify knowledge in a declarative fashion (symbolic encoding of the knowledge) provides considerable advantage in the manipulation of the knowledge over hard-coded subroutines in conventional programming.

According to Willians et al. [Will88], there are three main components in an ES execution cycle which is depicted in Figure 3-16:

- The *working memory* contains facts about the world which are either given or inferred at run-time. These facts are tested as the conditional parts of rules, and they can be asserted or deleted by rules as the tests succeed.
- The *rule memory* contains a number of rules that define the system's behaviour.
- The *interpreter*, also called the inference engine, selects rules from the rule memory and triggers them if the conditions match the content of the working memory.

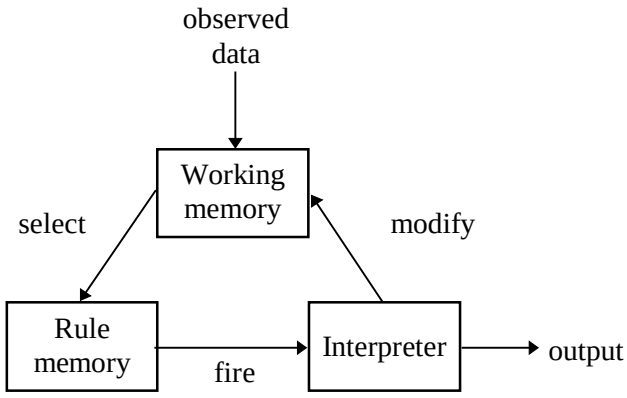

Figure 3-16: Expert system execution cycle.

3.8.2 Blackboard Systems

The complexity of the problems in certain applications and the increasing demands for KBS facilities in a wider variety of areas of applications have pinpointed limitations with the ESs. The BB model has emerged as a general model for implementing KBSs more comfortably than the ESs. It retains most of the features of the ES and adds to it a more structured organisation. It represents a major innovation over the traditional structure which is mostly based on search methods. Nevertheless, the BBM does not replace the ES in its original role. There is some overlap in their scope of application but the focus of the BBM is slightly shifted away from the profile of the ESs which are typically advisory or consulting systems.

Historically the idea of the blackboard originated from the HEARSAY-II system [Enge88]. HEARSAY-II is a speech-understanding system that recognises connected speech in a limited vocabulary. There, an organisational model was developed to

overcome the difficulties with ill-defined sources of knowledge. Thus, because of the uncertainty in recognising words directly from acoustic signals, a space of candidate solutions is created. By applying some strategic knowledge, only a few points in the solution space need be examined and developed before a solution is found. It was observed that such uncertainties are typical to many domain problems and that the problem-solving model was a quite general strategy to solve these problems.

The blackboard model is analogous to a group of experts with different skills that co-operate to solve a problem. Nii [Nii86] makes an interesting analogy with a hypothetical problem of a group of people in a room trying to put together a jigsaw puzzle on a large blackboard. At the start some people volunteer to place their most promising pieces. Each member of the group holds a number of pieces and sees if any of them fit into the pieces that are already on the blackboard. The whole process is effected silently without direct communication among the members. The analogy to the blackboard model can even be further extended. For example, suppose that there is a supervisor to monitor the access to the blackboard by the members because of the layout restrictions of the room such as a centre aisle which allows only one person to come through to the blackboard at once.

A typical blackboard model is described by the diagram in the Figure 3-17 [Nii86]. Basically it consists of three major components: *knowledge sources*, *blackboard data structure*, and *control*.

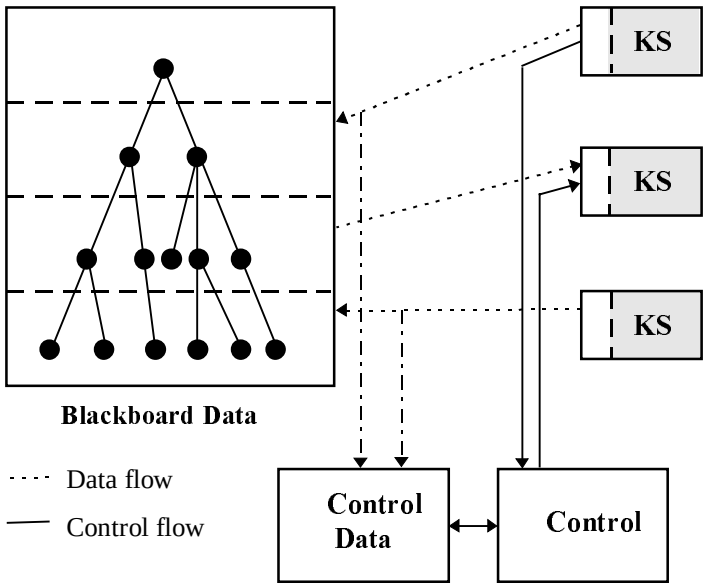

Figure 3-17: Nii's Generic Blackboard Framework.

The knowledge employed in the solution of a domain problem is partitioned into knowledge sources. Each knowledge source (KS) is a specialist that solves one aspect of the problem in a domain (analogous to members of a group of experts in the previous example). These KSs are invoked whenever a process in the blackboard requires them to proceed, then they produce changes to the blackboard data structure that incrementally contributes to a solution. Similar to production rules, the KSs are composed of two parts: the preconditions and the body of action. The KS becomes active as the preconditions are matched (Figure 3-18). A KS is similar to a rule, however, the action performed by a KS body is implemented as procedures or as a set of rules.

Figure 3-18: The general format of a knowledge source.

The *blackboard data structure* is a shared global data structure of the system that is accessed by the knowledge sources. It is also analogous to the (unstructured) working memory of the expert systems. It contains information about the state of the world and the partial solution effected by the KSs.

The *control* constantly monitors the blackboard context and decides which knowledge sources should be called in at each step of the problem-solving and schedules them for execution. This component is analogous to the chaining mechanism in the production system. The solution of a goal generally requires successive application of KSs. Whether the goal is solved in top-down or bottom-up order will depend on how the knowledge is structured in the blackboard.

The description of a blackboard model is quite flexible and permits different implementations ranging from simple to very complex [Nii86, Haye85]. The complexity of the design of blackboard systems in specifying and combining each of the three components depends to a great degree on the nature of the application problem itself. The blackboard model is an alternative reasoning control model for those problems that

production rules alone cannot solve properly. In essence the blackboard subsumes the production rules in a structured way and one of the most important features of the blackboard model is its capability to "discipline" the control mechanism.

Penny Nii notes the problem with the flexible definition of the blackboard model:

> "The difficulty with this description of the blackboard model is that it only outlines the organizational principles. For those who want to build a blackboard system, the model does not specify how it is to be realized as a computational entity, that is, the blackboard model is a conceptual entity, not a computational specification. Given a problem to be solved, the blackboard model provides enough guidelines for sketching a solution, but the sketch is a long way from a working system. To design and build a system, a detailed model is needed."

3.9 Summary

In this chapter we have reviewed the concept of *action* which is described as a transition between *states*. Simple actions are accomplished by simple *operators* while more complex (abstract) actions are accomplished by a sequence of operators. The problem of finding such a sequence of operations is called *planning*. Different actions and situations normally require different sequences of operations. Knowledge-based systems are AI programs that reason about the application domain. They are characterised by two components: knowledge representation and reasoning methods. The most general model for implementing knowledge-based systems is the blackboard as its structure can be effectively organised in a way that reflects the nature of the application problem.