Chapter 2

Intelligent agent

Software agents can be viewed as software entities designed to perform a designated function on behalf of a user/its human counterpart. For example, an accelerator controlling agent can be viewed as the virtual operator that controls the accelerator operations on behalf of the actual operator.

2.1 Agents as intelligent objects

The idea of the agent was first conceived by John McCarthy and the term was coined by Oliver G. Selfridge in 1950[52]. Their view of an agent was that of a system which, given a goal, could carry out the details of the appropriate computer operations and could ask for and receive advice, offered in human terms, when it was stuck. An agent would be a soft robot living and doing its business within the computer world [52]. Since then, the agent concept has been studied by many researchers from varied fields such as e-commerce [89], web based applications [75], distributed computing [86], medical diagnosis [37], supply chain distribution [71], production scheduling [1], industrial process control [80] and many more. To distinguish the intelligent agents from classical programs, different researchers have given different definitions.
Jennings and Wooldridge [45] define an intelligent agent as “a computer system that is capable of flexible autonomous action in order to meet its design objectives”. By flexible, they mean that the system must be responsive, proactive, and social. Agents should be able to perceive their environment and respond in a timely fashion to changes (responsive). Agents should be opportunistic or goal-directed when they respond to changes (proactive). Agents should also cooperate with other agents or humans in order to solve their problems and achieve the system goal (social). By autonomy, they mean that the system should be able to act without the direct intervention of humans or other agents, and should have control over its own actions and internal state.

Nwana and Ndumu [84] define an agent as “a component of software and/or hardware that is capable of acting exactingly in order to accomplish tasks on behalf of its user”. Parunak [85] defines intelligent agent as “an active object with initiative”, the next extended step to object-oriented programming in software evolution. While objects are passive and gain control only when some external entity sends them a message, agents can initiate actions and maintain control over localized code and data.

Hayes [34] defines an agent as “an entity that is capable of carrying out goals, and is part of a larger community of agents that have mutual influence on each other”. She emphasizes that partial autonomy and being part of a community are distinguishing properties of agent-based systems. Autonomy gives the system robustness and modularity and being part of a community makes it possible to build organizations of agents whose net effect is greater than the sum of their parts.

Bradshaw [14] enumerates attributes that agents might possess, such as reactivity, autonomy, collaborative behaviour, knowledge-level communication ability, inference capability, persistence of identity and state over long periods of time, adaptivity (being able to learn and improve with experience), and mobility.
From the above stated authors’ views, a concise definition of an intelligent agent can be formulated as an agent being an encapsulated software system situated in some environment, and developed with the notion of rational decision making in choosing their actions in an autonomous manner in this environment in order to maximise its chances of success towards achieving its goals. Further, intelligent agents must possess some of the attributes such as

- capable of acting in an environment
- able to communicate directly with other agents
- is driven by a set of tendencies (in the form of individual objectives or of a satisfaction/survival function which it tries to optimise)
- has resources of its own
- capable of perceiving its environment (but to a limited extent)
- has only a partial representation of its environment (and perhaps none at all)
- possesses skills and can offer services
- can improve its skills through learning
- tends towards satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, its representations and the communications it receives.
- capable of taking initiatives to adapt in dynamic environment

The realization of intelligent agents is mainly done by encapsulating the above stated attributes in modules and then these modules serve as the building block for the overall realisation of the agent structure.

Rzevski [97] and Shen *et. al.* [105] have summarised the minimum set of such modules needed for realising the intelligent agents as perception, cognition (or reasoning)
and execution (action). Shen et. al. [105] proceeds further to identify additional modules that may be included within the internal structure of an agent:

- communication interface
- social knowledge
- self knowledge (self representation)
- domain knowledge (domain representation)
- knowledge management
- learning
- problem solving methods
- co-ordination
- planning and scheduling
- control
- conflict management
- application interfaces

For our intelligent agent representation, we will follow that the model of perception, cognition and execution is a valid generic representation for any agent since the items in the above list could quite easily be grouped under these three headings.

### 2.2 Intelligent agent architectures

From conceptual abstract notion of intelligent agent to concrete realization of the actual intelligent agent, agent architectures play important role. Architecture of an agent refers to the internal organisation and interconnection of the constituent modules required to implement the agent behaviour. Agent architectures are thus linked to agent type and may be classified by behaviour or alternatively by the type of organisation structure.
Based on the kind of representation and reasoning used, many different types of agent architecture descriptions appear in the literature. Some of the widely recognised and well studied architectures are briefly described below.

### 2.2.1 An abstract agent

Starting from the agent definitions that the agents are situated in an environment, are capable of observing this environment through sensors, can work on this environment through some actuators and they posses some minimum attributes like perception, cognition and execution, the block diagram for such an abstract agent is shown in figure 2.1. Here the block *see* performs the function of percept preparation i.e. the preparation of system variables from the sensors data that can serve as agent/environment state defining variables (or at-least assist in identifying the system states). The block *next* plays the role of providing the cognitive behaviour to the agent through action deliberations and block *action* represents the agent’s mechanism for altering the environment through actuators.

Let $S$ be the set of environment states, $S = \{s_1, s_2, \ldots\}$, $A$ be the set of actions, $A = \{a_1, a_2, \ldots\}$. Now, if the sequence of agents interaction with the environment is pro-
vided as history, i.e, the sequence of state-action pairs.

\[ h : s_0 \overset{\hat{a}_0}{\rightarrow} s_1 \overset{\hat{a}_1}{\rightarrow} s_2 \overset{\hat{a}_2}{\rightarrow} \ldots s_{i-1} \overset{\hat{a}_{i-1}}{\rightarrow} s_i \overset{\hat{a}_i}{\rightarrow} \ldots \]

where \( s_0 \) is the initial state and \( \hat{a}_i \) is the action the agent performs when it is in the state \( s_i \). Now for any \( h \) to be a possible history of an agent starting from initial state \( s_0 \), all actions are generated from state sequence.

\[ \forall i \in N, \hat{a}_i = action((s_0, s_1, \ldots, s_i)) \]

and every new state must belong to the set of possible environmental states reachable from the previous state by applying the selected action.

\[ \forall i \in N, i > 0, s_i \in env((s_{u-1}, \hat{a}_{u-1})) \]

now for a purely reactive agent (i.e. the agent which decides its next action to be performed without referring to its history) will be described by the set of actions situation pairs given as.

\[ action : S \rightarrow A \]

where there is at-least one action mapping present from each state in \( S \) to actions in \( A \).

In a more formal way, the function \( see \) will be a mapping of system states to percepts.

\[ see : S \rightarrow P \]

and the function \( action \) will be a mapping from current percept to the actions.

\[ action : P^* \rightarrow A \]

and the function \( next \) is the system state updating function
Algorithm 1: Execution steps of an abstract agent.

Data: observed environment state(s : S), action set A
Result: perform action ˆa

initialise internal state to i_0;
repeat
  /*observe environment for state */
  READ s;
  /*and generate perception*/
  p ← see(s);
  /* select action according to internal state and percept */
  ˆa ← action(next(i_0,p));
  /*perform selected action*/
  DO ˆa;
until stop;

next : P* × P → P*

The execution cycle for such an abstract agent is shown in Algorithm 1

2.2.2 Subsumption agent architecture

Subsumption architecture falls in the category of reactive architectures where agents merely react to situations and do not reason about the world. Usually, both, the agents and the actions are relatively simple and global properties are seen as emerging from the interaction of behaviours [10, 39, 93]. The advantage of this approach is a faster agent response to changing environmental conditions so long as they have a predefined stimulus-response-pairing.

Subsumption architecture was first introduced by Brooks [11] for reactive agents and is mainly motivated by the following facts

- Intelligent agents can be designed without encapsulating the decision-making based on syntactic manipulation of symbolic representations of knowledge.
- The rational behaviour cannot be disembodied but is a product of the interaction the agent maintains with its environment.
The complex intelligent behaviour emerges from the interaction of various simpler behaviours.

It is a modular architecture with horizontal linking between modules as shown in figure 2.2, here the modules are organised in vertical layers. The modules operate in parallel, with those higher up in the organisation having a dominance over those lower down. This means that the higher modules can inhibit the behaviour of lower level modules. As with the modular architecture, the designer defines the connections between modules and the dominance relationships that exist between them in the form of inhibit rules. Usually the implementations of inhibiting relationship is implemented by means of priority assignment for individual behaviours.
Algorithm 2: Pseudocode for action selection by a subsumption agent.

Data: percept \((p : P)\), action set \(A\)

Result: selected action \(\hat{a}\)

/* get the list of all the rules for the percept \(P\) */
initialise fired:\(P(R)\);
/* for the current percept \(p\) generate the list of fired rules */
fired:=\{(c, \hat{a})|(c, \hat{a}) \in R\) and \(p \in c\};
/* if fired rule list is not empty */
if \(|\text{fired}| > 0\) then
  /* then select the highest priority rule */
  find minimum \((c, \hat{a})\);
  /* and return the action sequence associated with it */
  return \(\hat{a}\);
else
  /* otherwise do nothing */
  return \(null\);
end

Here the agent’s decision-making is realized through a set of task accomplishing behaviours. Each behaviour is like an individual action function, continually taking perceptual input and mapping it onto an action to perform. No complex symbolic representations and no symbolic reasoning are utilised at all. It mainly implements the rules to map \textit{situation} \(\rightarrow\) \textit{action} relationship. The percepts block accepts input from sensors and produces a set of percepts \(P\). The action is realised through a set of behaviour rules \(R\), together with an inhibition relation \(<\), over time where \(c\) is a set of percepts called that \textit{condition} and \(\hat{a}\) is an action.

\[
R = \{(c, \hat{a})|C \subseteq P, \hat{a} \in A\} \tag{2.1}
\]

A behaviour will fire in state \(s\) if some function \(see(s) \in c\) (if the condition is satisfied by the percepts). The inhibition relation is a total ordering on the behaviour rules. If \(r1\) inhibits \(r2\), then the inhibit rule can be written as \(r1 < r2\), i.e., \(r1\) is lower in the hierarchy than \(r2\) and hence will get priority over \(r2\), where \(r1, r2, \ldots\) are the rules (elements of \(R\)).
The subsumption architecture has been successfully used in robotic applications e.g. AGVs (automated guided vehicles) [126] and in control of Intelligent Geometry Compressor [76, 77] and many more, proving its significance. Reactive architectures like subsumption, on one hand are advantageous for being modular, supportive towards iterative development and testing of real-time systems in their target domain, and emphasising on connecting limited, task-specific perception directly to the expressed actions that require it. But, on the other hand, they suffer from drawbacks of inability to have many layers, since the goals begin interfering with each other, the difficulty of designing action selection through highly distributed system of inhibition and suppression, thus providing rather low flexibility at runtime.

2.2.3 Logic based agent architectures

Logic based architecture discussed by Genesereth and Nilsson [30] and others [62, 69, 95] realises the agents using the traditional approach to building artificially intelligent systems. It is based on the notion that intelligent behaviour can be generated in a system by giving that system a symbolic representation of its environment and its desired behaviour and syntactically manipulating this representation. This syntactic manipulation corresponds to logical deduction. In logic based architectures, the system’s symbolic representation is stored as a database of formulae of classical first-order predicate logic similar to a Prolog [20] database. This database mimics the information that agents have about their environment and plays a somewhat analogous role to that of belief in humans. The agent behaviour is mainly decided by the deduction rules.
Algorithm 3: Pseudocode for action selection by a logic based agent.

Data: system state $\Delta$, logic sentences set $L$, L-formulae set $D$, action set $A$
Result: selected action $\hat{a}$

/*get the deduction rules $\rho$ from the Knowledge-Base $K$ */
initialise $\rho$;

/*for each action $\hat{a}$ element of set $A$*/
for each $a \in A$ do
    /*if from the internal state $\Delta$ the formula $\varphi(\hat{a})$ can be proved using deduction rule $\rho$ */
    if $\Delta$ $\vdash$ $\rho$ $\varphi(\hat{a})$ then
        /*then return action $\hat{a}$ as the valid action*/
        return $\hat{a}$;
    end
end

/*for each action $\hat{a}$ element of set $A$*/
for each $a \in A$ do
    /*if from the internal state $\Delta$ the formula $\neg\varphi(\hat{a})$ can not be proved using deduction rule $\rho$ */
    if $\Delta$ $\not\vdash$ $\rho$ $\neg\varphi(\hat{a})$ then
        /*then return action $\hat{a}$ as the valid action*/
        return $\hat{a}$;
    end
end

/*otherwise do nothing*/
return null;

Let $L$ be the set of sentences of classic first-order logic and $D = \varphi(L)$ be the set of sets of L-formulae. Now if $\Delta, \Delta_1, \Delta_2, ...$ are the members of $D$ representing the agent’s internal state, then the agent’s decision making process is modelled through a set of deduction rules denoted by $\rho$. The choice of action by the agent is decided through proving the formula $\varphi(\hat{a})$ by theorem proving process based on the present database representing the system state. i.e. if $\Delta$ $\vdash$ $\rho$ $\varphi(\hat{a})$ or $\Delta$ $\not\vdash$ $\rho$ $\neg\varphi(\hat{a})$ can be proven from an internal state $\Delta$ using only the deduction rule $\rho$ then the action $\hat{a}$ is the valid action for applying to the system. The algorithm for action identification for such an agent is given in Algorithm 3.

Along with the logic based action selection mechanism, the agent will have the normal
blocks for mapping the functions from sensors to the percepts in a more elaborative approach through symbolic representation. Thus the complete execution cycle will be given by.

\[
\text{see} : S \rightarrow P \\
\text{next} : D \times P \rightarrow D \\
\text{action} : D \rightarrow A \]

where the functions see, next, action are the basic function of abstract agent.

Using logic based approach, the agent’s behaviour can be guaranteed. This may be useful for safety-critical applications or applications of high priority such as tackling those parts of accelerator control which can directly result into beam killing by the immediate wrong agent actions rather than simply degrading the operation performance. The theorem proving process takes time and by the time the agent proves which action is optimal, the environment may have changed. This leads to the problem of calculative rationality (i.e. the decision making apparatus produces action that was optimal when decision making process began ). Therefore this type of architecture is only suitable when environment doesn’t change faster than the agent can make decisions. Also, many times, the representation of procedural knowledge and reasoning about temporal information is not easy for implementation point of view.

### 2.2.4 Deliberative (Beliefs-Desire-Intention) architecture

BDI agent model presents the abstraction of rational agent based on the notion of belief-desire-intention, inspired by the human practical reasoning and decision process i.e. the process of deciding, moment by moment which action to perform in the furtherance of desired goals. For this, the reasoning by humans involves two important phases: (1) what goals a human wants to achieve and (2) how he is going to achieve them. The first
is also called the *deliberation* process and the latter is called *means – end reasoning* [12, 13, 31, 98, 99]. The architecture of BDI agent is outlined in the block diagram in figure 2.3. There are seven main components in a BDI agent.
Note: – Let Bel, Des and Int denote large abstract sets from which beliefs, desires and intentions can be taken. The state of a BDI agent is at any moment a triple \((B, D, I)\) where \(B \subseteq \text{Bel}, D \subseteq \text{Des} \text{ and } I \subseteq \text{Int}\).

1. A set of beliefs representing information the agent has about its current environment \(\wp(\text{Bel})\).

2. A belief revision function, \((brf)\), which takes a perceptual input and the agent’s current beliefs, and on the basis of these, determines a new set of beliefs. Therefore it is a mapping from a belief set and percept into a new belief set

\[ brf : \wp(\text{Bel}) \times \mathcal{P} \rightarrow \wp(\text{Bel}) \]

3. An option generation function, \((\text{options})\), which determines the options available to the agent \(\text{(its desires)}\), on the basis of its current beliefs about its environment and its current intentions; thus it maps a set of beliefs and a set of intentions to a set of desires.

\[ \text{options} : \wp(\text{Bel}) \times \wp(\text{Int}) \rightarrow \wp(\text{Des}) \]

Here, the main function of options is means – end reasoning, and this must be consistent with beliefs and current intentions as well as opportunistic to recognise when environmental circumstances change advantageously.

4. A set of current options, representing possible courses of actions available to the agent.

5. A filter function \((\text{filter})\), which represents the agent’s deliberation process, and determines the agent’s intentions on the basis of its current beliefs, desires, and intentions.

\[ \text{filter} : \wp(\text{Bel}) \times \wp(\text{Des}) \times \wp(\text{Int}) \rightarrow \wp(\text{Int}) \]
6. A set of current intentions, representing the agent’s current focus i.e. the goals it has committed to trying to bring about.

7. An action selection function (execute), determines an action to perform on the basis of current intentions.

\[
execute : \wp(\text{Int}) \rightarrow A
\]

Desire and intention are the mental attitudes concerned with the actions and beliefs representing the agent’s information and knowledge about its environment.
Algorithm 4: Standard BDI interpreter.

Data: Sets beliefs $B$, Desires $D$, Intentions $I$

Result: perform action()

initialise internal state;

repeat

/* observe environment for events */
READ event – queue;
/* and generate options */
options ← OPTION-GENERATE(event – queue);
/* select valid options */
selected – options ← DELIBERATE(options);
/* add the selected options to intentions list */
UPDATE-INTENTIONS(selected – options);
/* execute one action execution cycle */
EXECUTE(intentions);
/* get new external events */
READ event – queue;
/* remove the sucessful attitudes */
DROP(successful-attitudes);
/* remove the impossible attitudes */
DROP(impossible-attitudes);

until stop;

In BDI agent implementations the abstraction of desire is represented by the set of goals for the agent. An active goal can be considered as the desire that has been adopted for active pursuit by the agent. For achieving the goals, the agent implementation construct provides mechanism for defining different plans. A plan is an ordered list of actions which must be performed for achieving the goal. The intentions represent the current plans the agent has chosen to execute or is currently executing. Figure 2.4 shows the block diagram of a BDI agent implementation [66]. The algorithm 4 shows the classical BDI interpreter proposed by A. S.Rao and M.P.Georgeff [91].

At the beginning of each cycle, the option generator reads the event queue and returns the list of options. Options are basically the goals that the agent perceives viable for consideration at an instant of time. From implementation point of view, options can
be viewed as the list of all the alternate plans that are meant for achieving a particular goal. Each plan comprises of a body that describes the sequence of actions or sub-goals that have to be achieved for plan execution to be successful. The conditions under which a plan can be chosen as an option are specified by an initiation condition (triggering events) and a precondition, specifying the situation that must hold for the plan to be executable.

Next, the deliberate procedure selects from the options list those plans which are found suitable for immediate execution and lists them as selected-options. The update-intentions procedure then updates the agent’s beliefs for the immediately selected intentions. The execute procedure carries out the actual work of acting upon the environment towards achieving the intended goal by sequentially executing the recipe specified in the body of the selected plan. The next three procedures get-new-external-events, drop-successful-attitudes and drop-impossible-attitudes basically update the event queue with the system generated events and external events and filter out the un-relevant goals before proceeding for the next interpreter cycle.

Figure 2.5: The Layered agent architecture (a) horizontal layering, (b) vertical layering with one pass and (c) vertical layering with two pass.
2.2.5 Hybrid architectures

Hybrid architectures also called as hierarchical architectures [28, 73] attempt to combine both reactive architecture and deliberation architecture in order to take advantages of both. As a result such architectures are capable of providing fast reactive behaviour as well as more intuitive cognitive behaviour by incorporating the learning and sustained, pro-active, goal directed behaviour. In the layered architecture, the various subsystems of the agent capable of providing reactive and pro-active behaviours are arranged in a layered structure.

Often, the reactive component is given some kind of precedence over the deliberative one, so that it can provide a rapid response to important environmental events. The Touring Machines [28], INTERRAP [73, 74], and CIRCA [72] are good examples of such architectures. Normally in an agent body subsystems are arranged into a hierarchy, with higher layers dealing with information at increasing levels of abstraction. Typically, there will be at least two layers, to deal with reactive and pro-active behaviours respectively.

Two types of control flow mechanisms are normally used within layered architectures

1. Horizontal Layering,

2. Vertical Layering.

Horizontal Layering

In horizontally layered architectures, the software layers are each directly connected to the sensory input and action output. This is shown in figure 2.5(a) This approach is simple and implements agent by implementing $n$ layers, for $n$ different types of behaviour. To overcome the problem of interfering of one layer’s action with that of other layer, when layers compete with one another to generate action suggestions, central control is exercised through mediator function, where mediator decides about the layer to which the task of
control is given. This forms a bottleneck when the number of layers increases as the mediator function has to consider all possible interactions between layers before deciding about the layer of the agent that should take control.

**Vertical Layering**

In vertically layered architectures, (figure 2.5(b),(c)), sensory input and action output are each dealt with by at most one layer each. This type of architectures are classified into one pass architectures and two pass architectures. In one-pass architectures, control flows sequentially through each layer, until the final layer generates action output. In two pass architectures, information flows up the architecture and control then flows back down. Both these architectures reduce the complexity of interaction between layers thus relaxing the control bottleneck problem faced in the horizontal layering but at the same time suffers from the problem of fault intolerance, as failure in any one layer can seriously affect the agent performance.

### 2.2.6 Modular architecture

This type of architecture is widely used in multi-agent systems and may range from very simple, comprising a few modules to complex organisations involving a large number of modules. It is sometimes referred to as a horizontal-module architecture since the modules are at the same level in the organisation. Also, in this type of architecture, all of the connections between the modules are typically fixed i.e. the information flow is pre-defined by the agent designer. The simple example shown below reveals the basic perception, cognition and execution structure. Figure 2.6 shows the different modules of such an agent system. A few examples of modular architecture are robotic applications[51], network diagnostics [3] and in service agents[78].
Figure 2.6: The modular agent architecture.

### 2.3 Multi agent systems

The agent based problem solution and exercising of system control is extended further by designing the system comprising of multiple agents where each agent is directed towards achieving its local goal and contributes its share in achieving the common goal of the community. The field is covered under the concept of Distributed Artificial Intelligence (DAI) where the agents are grouped together to form communities which cooperate and interact to achieve the goals of individuals and of the system as a whole. Further, it is assumed that each agent is capable of a range of useful problem-solving activities in its own right, has its own aims and objectives and can communicate with others [41]. The multi-agent based approach requires the enhancement of individual agent behaviours/capabilities irrespective of the way they are implemented and of the method/methodology adopted by underlying agent implementation for participating as an individual in an multi-agent system. Some of these capabilities are communication, coordination, cooperation and collaboration as discussed in the following.
Communication

In multi-agent systems the mechanism to communicate between different agents working as a team is a must for information exchange. This information exchange can be accomplished through environment in an indirect way in which one of the agent modifies the environment with an intention that following agents will capture the purposefully introduced disturbance in the environment and will infer the message to be conveyed. This type of primitive biologically inspired information exchange is of limited use but is an important trick of communication through behaviour and mainly employed for physical agents like robots with very simple structures and limited resources [29]. The direct communication mechanism is used when the agents are situated in an networked environment. The two main types of communication methods of this type are shared memory and message passing. The most widespread example of the former is the blackboard system [26, 27], where the blackboard is a global database containing entries generated by the agents. The entries include intermediate results generated during problem solving and include both elements of the problem solution and information deemed important in generating solution elements. Message passing ideas have been drawn from conventional object-oriented programming and in particular from object-based concurrent programming and is the widely adopted approach for inter-agent communication [42, 53, 119]. Message passing has some advantages over the blackboard system. In particular, shared memory systems generally do not scale up well - a single blackboard can be a severe bottleneck and multiple blackboards have the same semantics as message passing systems.

At the extreme of sophistication is the use of formal languages involving extended exchange of series of messages to support a conversation between agents [117]. In this context there is much, and growing interest in the field of ontology as a possible mechanism for agents to share the meaning of exchanged symbols [105].
Co-ordination

Co-ordination may be regarded as the process by which an agent reasons about its local actions and the (anticipated) actions of others to try and ensure that the community acts in a coherent manner, to achieve the overall goals of the system. Without coordination, the benefit of decentralized problem solving vanishes and the community may quickly degenerate into a collection of chaotic, in-cohesive individuals. Further, the co-ordination process ensures that all necessary portions of the overall problem are included in the activities of at least one agent. Specific examples of coordination activities include supplying timely information to needy agents, ensuring the actions of multiple actors are synchronized and avoiding redundant problem solving [67]. Various co-ordination techniques have been devised like organisational structuring, subcontracting, negotiation and multi-agent planning. Basically all of these techniques use the following mechanisms for co-ordination at one stage or the other in their operation cycle.

- *Mutual adjustment* - agents share information and resources to achieve some common goal by adjusting their behaviour according to the behaviour of the other agents,

- *Direct supervision* - one agent has some degree of control over others which may have been arrived at through mutual adjustment,

- *Standardization* - supervisor agent establishes standard procedures for agents to follow in given situations,

- *Mediation* - one agent serves as a facilitator or broker to influence interaction between agents.
Collaboration

Collaboration arises when one agent is able to perform a task, which only it can do, and as a result enables another agent to achieve its own goal. Clearly, the need for collaboration is determined by the allocation of skills and resources to agents made when the system was designed. Except in simple cases, it is often necessary to coordinate collaboration in order to make effective use of skills and resources consistent with overall system goals. The process of deduction which agents have to collaborate for the given goal is governed by the coalition formation strategies. A number of coalition formation algorithms have been developed to determine which of the potential coalitions should actually be formed [23, 55, 83, 100]. All of these coalition formation processes include three main activities [101].

- **Coalition structure generation**- partitioning the set of agents into exhaustive and disjoint coalitions. Such a partition is called a coalition structure.

- **Optimizing the value of each coalition**- pooling the tasks and resources of the agents in every coalition, in the coalition structure, in order to maximize the coalition value.

- **Payoff distribution**- dividing the value of each coalition among its members so as to achieve stability or fairness.

Co-operation

Co-operation is about agent’s actions being mutually supportive to their respective goals. Supportive action by one agent for another may be intentional or incidental. Ferber [29] states that, put simply, the problem of co-operation condenses down to determining who does what, when, by what means, in what way and with whom. Ferber summarises this in the formula:
Co-operation = collaboration + co-ordination of actions + resolution of conflicts

Ultimately, the co-operation strategy of a multi-agent system is critical in ensuring that actions by autonomous agents in pursuit of local goals have, at least, a beneficial, if not optimal, effect on overall system performance.

2.4 Intelligent control in accelerator scenario

In accelerator community the first use of agent application was demonstrated by Jennings [41]. Jennings connected the two diagnostic expert systems in the accelerator control environment namely CODES(control system diagnosis expert system) and BEDES (beam diagnosis expert system) by using the ARCHON framework, thus converting them to two distinct intelligent agents, which can work together to fulfil common goal of diagnosing the faults. The ARCHON framework was composed of four main components: a high-level communication module (HLCM), which manages inter-agent communication; a planning and coordination module (PCM), which is essentially responsible for deciding what the agent will do; an agent information management module (AIM), which is responsible for maintaining the agent’s model of the world and finally, an underlying intelligent system (IS), which represents the agent’s domain expertise. The HLCM, PCM and AIM together constitute a kind of ‘agent wrapper’, which was used to encapsulate the existing intelligent system to turn them into agents. Strengths of ARCHON architecture were: use of a top down approach to look at the overall needs of the application and a bottom up approach to look at the capabilities of the existing system so that the development efforts can be minimised. Problem solution was achieved through pre-specified recipes called as plans. Plans were represented in the form of tree with nodes as tasks and arcs as
conditions. The self model and the acquaintance model were provided for assistance. Apart from accelerator fault finding, the ARCHON architecture has been successfully applied to different other industrial applications [40, 44, 111, 119, 120] also primarily for assisting in the fault finding operation.

Another effort towards the intelligent control of accelerator was done by Klein et.al. [56, 57, 58, 59, 60, 113]. They proposed a control system architecture based on hierarchical, distributed, knowledge-based controllers, each of which is an expert in controlling some section of the environment or in performing some function over that environment. These Knowledge-based controllers were provided with the capabilities for planning, diagnosis, and learning, as well as knowledge acquired from human domain experts to select, sequence, and configure control actions. Controllers were also designated with the responsibility for reasoning about the system state, diagnosing errors in controller actions, decomposing goals into tasks and actions, and initiating human intervention as and when found necessary.

Means for implementing the general purpose optimization or control algorithms, such as hill-climbing optimization, fuzzy logic or neural network-based feedback control, conventional control loops, etc were provided through services called as solvers. More complex procedures were formulated by coordinated assembly of individual solvers. An object oriented physical access layer (PAL) was developed as an abstraction mechanism between controllers and the underlying control system to provide a mechanism for hiding unimportant implementation details about the domain hardware and provide a uniform interface for control access.

A new control approach called as teleo-reactive(TR)[81] control was utilised. Which is a combination of feedback-based control and traditional discrete action planning. TR programs sequence the execution of actions that have been assembled into a goal-related
plan. Here, unlike traditional planning environments, no assumption is made that actions are discrete and un-interruptible and that an action’s effects are completely predictable. On the contrary, teleo-actions are executed as long as the action’s preconditions hold and its goal has not yet been achieved (unless some other action closer to the final goal becomes activated). A short sense-react cycle ensures that when the environment changes, the control action changes to fit the new state. The strongest point of this approach is that the TR tree execution is adaptive in that, if some unanticipated event in the environment reverses the effects of previous actions, TR execution will typically fall back to some lower level condition and restart its work towards the top level goal. On the other hand, if something good happens, TR execution is opportunistic i.e. when some higher condition unexpectedly becomes true, execution shifts to the action associated with that condition.

In addition to the above stated accelerator control efforts, there are two more examples which although do not come under intelligent agent based control but need their mentioning as these are related to the scope of this thesis. The first is related to the application of intelligent system concepts to automatic beamline alignment problem. Pugliese et.al.[90] proposed a model reference based architecture by combining the hierarchical and subsumption architecture. The strength of this approach is the coupling of an artificial intelligence and a real-time sub-system as parallel, cooperating components. The second is related to the electron transport line optimization using neural networks and genetic algorithms, where Schiemer et.al.[102] proposed the accelerator (particularly the transport line) tuning using the artificial intelligence based scheme.

ARCHON architecture although talks about the use of self model but lacks the ability of adaptation mechanism in it which may be required for achieving the close loop control objective. Further to this, the architecture addressed all the system parameter setting operations/system parameter reading operations through the AL-IS Interface (which is
acceptable for the diagnostic applications and is a must to make the application platform independent) but for implementation of local closed loop control, the methodology for direct transactions with system inputs/outputs can serve as a better alternative.

The TR based architecture proposed by Klein et.al. solves the lack of adaptation and I/O operation through intermediate layer problem of ARCHON and is thus more suited for the accelerator tuning and control application. But it experiences a problem of execution failure of TR tree under certain conditions, for example, if the TR tree enters a state in which no node is active, or if the TR tree is stuck in a cycle involving two or more nodes, or if the tree is stuck in a cycle involving a single node. The first case arises because the control plan is incomplete and does not include an action responding to some unexpected state. In the second case some node repeatedly terminates execution before its parent node becomes active (because it makes its own pre-image condition false). In the third case a single node repeatedly executes without making its parent active. The solution to these requires automatic derivation of action models from accelerator models. Further to this, this approach requires the intelligence at the controller level which may be possible to achieve for new machines to be developed, or for machines which are under the process of up-gradation, whereas it may not always be possible to incorporate the changes at controller levels for the existing machines.

2.4.1 ARCHON framework

ARCHON stands for ARchitecture for Cooperative Heterogeneous ON-line systems. It devised a general-purpose architecture, software framework, and methodology which has been used to support the development of distributed artificial intelligence (DAI) systems in a number of real world industrial domains [40, 44, 111, 119, 120]. The ARCHON framework focusses on the real implementation issues rather than the symbolic representation
and semantics of the agent based control problem. The methodology is mainly focussed on the distributed problem solving using multi agent scenarios and tries to address both types of real word systems, purpose-built and pre-existing systems, where ARCHON basically acts as the gluing platform which joins the distributed intelligent systems (IS) capable of exhibiting some semi-autonomous behaviour and extends their ability towards joint problem solving for common goal. In ARCHON, individual problem solving entities are called agents. These agents have the ability to control their own problem solving and to interact with other community members. The interactions typically involve agents cooperating and communicating with one another in order to enhance their individual problem solving and to better solve the overall application problem. Each agent consists of an ARCHON Layer (AL) and an application program (IS) as shown in the figure 2.7.

The ARCHON's modular and layered implementation architecture comprises of mainly four modules - Monitor, Planning and Coordination Module (PCM), Agent Information Management (AIM) module, and High Level Communication Module.

**Monitor**

The Monitor is responsible for controlling the local IS. Each IS task is represented in the Monitor by a monitoring unit (MU). MUs present a standard interface to the Monitor whatever the host programming language and hardware platform of the underlying IS could be. These MUs can send and receive messages (of type *directives*, *confirmations* and *requests*) to and from the IS. All messages pass through the AL-IS interface which performs the translation and interpretation required for the IS to understand the AL directives and for the AL to understand the IS messages. For the IS to be able to react to an AL directive, the interface translates the command into the corresponding local control action. However, the interpretation and implementation of commands at IS are
left to the IS implementer domain.

MUs represent the finest level of control in the AL, and at the next level of granularity there are plans. Plans are pre-specified, acyclic, OR-graphs, in which, the nodes are MUs and the arcs are conditions. These conditions can: be dependent on data already available from previously executed MUs in the plan, be dependent on data input to the plan when it started, make use of the locking mechanism for critical sections of the plan, or be used to return intermediate results before a plan has completed. The plan mechanism is provided with inbuilt backtracking facility which can be used to express preferences and deal with complex alternatives.

The highest level at which the IS’s activities are represented is the behaviour level.
Behaviours contain a plan, a trigger condition for activating the behaviour, descriptions of the inputs needed by the activity and the results which will be produced, and any children of the behaviour. There are two types of behaviour: those that are visible to the PCM (and the other AL components) and those that are purely internal to the Monitor. The former type are called *skills* and they may be triggered by new data (either arriving from other agents or which the agent has generated itself) or by direct requests from other agents.

**Planning and Coordination module**

The PCM is the reflective part of the AL, reasoning about the agent’s role in terms of the wider cooperating community [43]. It is composed of generic rules about cooperation and situation assessment which are applicable in all industrial applications - all the domain specific information needed to define individual behaviour is stored in the self and acquaintance models. The former contains information about the local IS and the latter contains information about the other agents in the system with which the modelling agent will interact. The type of information contained in both models is approximately the same, although it varies in the level of detail, and includes the agent’s skills, interests, current status, workload and so on. For example, in order to determine how to obtain information which is needed to execute a behaviour but which is not currently available, the PCM will make reference to its self model to see if the information can be provided locally by executing an appropriate skill. If the information cannot be provided locally then the acquaintance models are checked to see if another community member can provide it. When the Monitor gets some results from a behaviour, firstly, the PCM checks the self model to see if the data can be used locally and then it examines its acquaintance models to see if any other agents are believed to be interested in receiving the data. And finally the PCM deals with requests arriving from other agents. By reference to its self
model, it will decide whether to honour the request and will then activate the necessary skill to provide the requested data. When the information is available it will ensure that a reply is directed to the source of the request.

**Agent information management module**

The AIM module is a distributed object management system which was designed to provide information management services to cooperating agents [116]. Within ARCHON, it is used to store both the agent models and the domain level data. As an illustration of the agent models, consider an agent, which is capable of producing information about ALARM-MESSAGES. The interest slots of its acquaintance models contain those agents who are interested in receiving this information and the conditions under which they are interested (a null condition signifies in all cases). The following portion of the acquaintance model specifies that an agent called BRS is interested in ALARM-MESSAGES which contain chronological information, that an agent called AAA is interested only in non-chronological alarm messages, and that an agent called BAI is only interested in non-chronological alarm messages which have the string INT within their ALARMS field:

**INTEREST-DESCRIPTION**

**INFORMATION-NAME:** ALARM-MESSAGES

**INFORMATION-CONDITION:**

```plaintext
["BRS", (CONTAIN (ALARM-MESSAGES "CHRONOLOGICAL "YES""));
("AAA", (CONTAIN (ALARM-MESSAGES "CHRONOLOGICAL "NO""));
("BAI", (AND (CONTAIN (ALARM-MESSAGES "CHRONOLOGICAL "NO""))
(_CONTAIN (ALARM-MESSAGES.ALARMS "INT")))));
```
High Level Communication Module

The High Level Communication Module (HLCM) allows agents to communicate with one another using services based on the TCP/IP protocol. The HLCM incorporates the functionality of the ISO/OSI Session Layer which continuously checks communication links and provides automatic recovery of connection breaks when possible. Information can be sent to named agents or to relevant agents (decided by reference to interests registered in the acquaintance models).

The ARCHON architecture has been used to integrate a wide variety of application program types under the general assumption that the ensuing agents will be loosely coupled and semi-autonomous. The agents are loosely coupled since the number of interdependencies between their respective ISs are kept to a minimum; the agents are semi-autonomous since their control regime is decentralised (meaning each individual ultimately decides which tasks to execute in which order). The ISs themselves can be heterogeneous - in terms of their programming language, their algorithm, their problem solving paradigm, and their hardware platform. An AL views its IS in a purely functional manner, it expects to invoke functions (tasks) which return results, and there is a fixed language for managing this interaction.

There is no centrally located global authority and each agent controls its own IS and mediates its own interactions with other agents. The system’s overall objectives are expressed in the separate local goals of each agent in the agent community. Because the agent’s goals are often interrelated, social interactions are required to meet global constraints and to provide the necessary services and information. Such interactions are controlled by the agent’s AL; relevant examples include: asking for information from acquaintances, requesting processing services from acquaintances, and spontaneously volunteering information which is believed to be relevant to others.
In more detail, an agent’s AL controls tasks within its local IS, decide when to interact with other agents (for which it needs to model the capabilities of its own IS and the ISs of the other agents), and communicate with its acquaintances. It is these basic functionalities that ARCHON’s modular and layered architecture provides to the designer to help him transform the existing systems to intelligent agent.

2.4.2 Teleo-reactive (TR)control

Nils Nilsson proposed the Teleo-reactive agents [81] based on the notion of teleo-reactive programming for planning and action representations. Teleo-reactive agents [9, 50, 82] react to their perceptions of the world by obeying an internal program (or policy) mapping perceptions to actions. The simplest policy structure is a set of mutually-exclusive production rules of the form \( \text{perception} \rightarrow \text{action} \), usually intended to control durative behaviour: given some current perception the agent performs the corresponding action until acquiring a new perception, whereupon it reacts likewise to that. Such an agent may or may not have sufficient perceptive capability to know, at any instant, the entire state of the world. An agent of this kind presumes that, it is capable of perceiving an intended goal state, whenever that state arises, and is accordingly designed with that capability in mind. Its program includes an explicit test for the goal state, whilst the nature and ordering of its rules are inferred by reductive analysis of that test. Its goal-orientatedness is thus explicit in the program.

TR control occupies a region between feedback-based control and traditional discrete action planning. TR programs sequence the execution of actions that have been assembled into a certain kind of goal-related plan. Unlike traditional planning environments, no assumption is made that actions are discrete and uninterruptible and that an action’s effects are completely predictable. On the contrary teleo-actions are typically durative
and are executed as long as the action’s preconditions hold and its goal has not yet been achieved (unless some other action closer to the final goal becomes activated)[41]. A short sense-react cycle ensures that when the environment changes, the control action changes to fit the new state.

TR action sequences or plans are represented in a data structure called a TR tree. A TR tree can be described as a set of condition-action pairs:

\[
\begin{align*}
    c_0 & \rightarrow \hat{a}_0 \\
    c_1 & \rightarrow \hat{a}_1 \\
    \vdots \\
    c_n & \rightarrow \hat{a}_n
\end{align*}
\]

where \( c_s \) are conditions and \( \hat{a}_s \) are the associated actions. \( c_0 \) is typically the top level goal of the tree and \( \hat{a}_0 \) is the null action, i.e., do nothing if the goal is achieved. At each execution cycle the \( c_i \) are evaluated from top to bottom until the first true condition is found. The associated action \( \hat{a}_i \) is then performed. The evaluation cycle is then repeated at a frequency that simulates the reactivity of circuit based control. TR trees are constructed so that each action \( \hat{a}_k \), if continuously executed under normal conditions, will eventually make some condition higher in the tree true. This then ensures that under normal conditions the top level goal, \( c_0 \), will eventually become true. TR tree execution is adaptive in that if some unanticipated event in the environment reverses the effects of previous actions, TR execution will typically fall back to some lower level condition and restart its work towards the top level goal. On the other hand, if something good happens, TR execution is opportunistic: when some higher condition unexpectedly becomes true, execution shifts to the action associated with that condition. Figure 2.8 represents such a typical TR tree showing the relationships between goals, actions, and
sub-goals. At each execution cycle, the action associated with the highest true condition in the tree is selected for execution. When the highest active level contains more than one action with satisfied preconditions, some arbitrary probabilistic method is used for choosing between possible actions [81].

Figure 2.8: teleo-reactive tree graph[113].

Construction of TR trees can be accomplished through a planning algorithm. Starting from the top level goal, the planner searches over actions whose effect includes achievement of the goal. The preconditions of the action generate a new set of subgoals, and the procedure recurses. Termination is achieved when the preconditions of one of the leaf nodes of the tree is satisfied by the current state of the environment. That is, the planning algorithm regresses from the top level goal through goal reduction to the current state. Actions, of course, generally have side effects, and the planner must be careful to verify that an action at any level does not alter conditions that are required as preconditions of actions at a higher level. TR tree planning algorithms typically build plans whose leaf nodes are satisfied by the current state of the environment. They do not build complete plans, that is, plans that can start from any world state, because such plans would generally be too large to efficiently store and execute. This is an important point because sometimes an unexpected environmental event can shift the world to a state in which no action preconditions in a TR tree are satisfied. In such case, replanning is done.
2.5 Summary and conclusion

In this chapter the concept of intelligent object agent is introduced. From literature the important agent properties that distinguish agents from simple computer programs emphasised by researchers are discussed. Starting from the abstract agent representations, the important agent implementation architectures along with their merits and demerits are presented. Extending the agent based system scenario, the important parameters necessary for multi-agent based system implementation are introduced. The literature survey on the use of agent based technology and the intelligent control concepts in the context of particle accelerators is then presented with an extended discussion on the case study of ARCHON framework and teleo-reactive control.

In chapter 4 we will see that the accelerator control system is comprised of distributed and layered structure; thus for accelerator control environment the subsumption architecture being simple and fast, can serve as the best candidate for implementations at the lowermost layers, that, demands a fast reactive cycle but are having the limited computational power. Further, in the implementation of agents for such controllers, the subsumption architecture can be extended with the communication capability already available with the accelerator control environment, thus embedding the coordination mechanism to such agents that enhances their integrity with the multi-agent based overall control of the facility.

The modular nature of the accelerator control system can be effectively utilised by using the modular architectures for agents that are needed to be implemented at middle or uppermost layers of accelerator control system. These agents can be implemented with the modules that either directly embed the underlying control system interface into their perception and execution modules or can simply communicate with the subsystems through message passing thus decreasing the implementation and testing time. Further
to this, the concepts from BDI agent architecture could be mixed with the modular architecture for agent implementations with multi-level plan library for handling the systems dynamics.

The logical agent architecture could be utilised for implementing the agents at the higher layers where all the accelerator parameter information from different subsystems is available. These agents could assist the operators / underlying low complexity agents by providing the diagnostic information or by providing the guidance for deciding their actions for example the logical agents can inform to the local beam line tuning agents about the faulty device which it is using to correct the beam or it can provide the information about the effective pair of correctors for the particular accelerator operating condition that the agent can use for beam correction.

For accelerator environment on broader perspective, the overall accelerator control job can be subdivided among different agents of varying architecture that can be implemented on different accelerator control system layers of varying complexity and computation power. These agents can be logically integrated with sufficient granularity of work division and responsibility delegation to work as a single, multi-agent based control system that reduces the operator efforts in accelerator tuning and control job.