CHAPTER 6

TEST BED MODELLING AND CONSTRUCTION

6.1 INTRODUCTION

The construction of Multi-Agent Systems (MAS) offers a natural and exhilarating paradigm to develop sophisticated programs in dynamic and open environments, particularly in distributed domains. MAS allows the macro components of a problem domain to be subcontracted by different problem solving agents with anticipated functionalities. On the other hand, from the programmer’s perspective, the modularity of MASs can lead to simpler programming rather than tackling the whole task. Since they are inherently modular, it will be easier to add new agents to a MAS organization than to a monolithic system in addition to the improved fault tolerant ability of the overall system.

Even though it is not necessary to implement a MAS on multiple processors, in order to comprehend the factual merits of MAS, its agents should be distributed across heterogeneous environment. Our proposed MAS is constructed in such a way that it provides the significant merits of MAS at different levels. This chapter exclusively presents an expressive modeling and construction of the test bed, which allows the MAS designers to focus on the conceptual design, rather than on low-level system programming issues. In other words, here the focus of the MAS construction is completely oriented towards realizing a real time MAS environment in order to facilitate investigations in such a way to validate and justify the proposed metrics
models, rather than assessing the effectiveness of the proposed MAS in enhancing the performance of the intended application domain.

6.2 TEST BED MODELLING

A Multi-Agent based distributed testing environment (Dhavachelvan et al 2005) is chosen as the test bed for the research presented in this thesis. The second consideration in this study is the construction of a prototype testing situation to assess the software systems using different testing strategies under distinct environments. In this investigation, the job of software testing in terms of common testing techniques is viewed as an application domain and it is being implemented using versatile software agents. Here the aim of the investigation is well defined such that only to assess the MAS w.r.t. the proposed metrics models rather than estimating the performance of the application domain. The proposed MAS consists of three different types of agents; distributor agents, testing agents and cloning agents. The proposed system can be defined as follows:

**Definition - 6.1:** Let ‘S’ be the proposed MAS for providing variety of testing environments and it can be defined as,

\[
S = \left\{ (D_1, D_2, D_3, \ldots, D_x), \\
(a_1, a_2, a_3, a_4, \ldots, a_x), \\
a_1 = (a_{11}, a_{12}, a_{13}, \ldots, a_{1(k-1)}), \\
a_2 = (a_{21}, a_{22}, a_{23}, \ldots, a_{2(k-1)}), \\
\vdots \\
\vdots \\
\vdots \\
a_x = (a_{x1}, a_{x2}, a_{x3}, \ldots, a_{x(k-1)}) \right\}
\]

(6.1)

where,

- ‘D’ is the distributor agent and ‘z’ is the number of distributor agents in the MAS.
• ‘a\textsubscript{v}’ is the testing agent and ‘x’ is the number of testing agents in the MAS and then \(0 < v \leq x\). This multi-agent framework provides scalar type testing environment. i.e. one agent can provide the testing environment with only one testing technique. And hence ‘x’ is the number of testing agents and also the number of testing techniques available in the MAS.

• ‘ac’ refers to the cloning agent(s) and ‘\(ac\textsubscript{w}\)’ is one of the clones of ‘\(a\textsubscript{v}\)’ and then \(0 < w \leq k\textsubscript{v} - 1\).

• ‘\(K\textsubscript{v} - 1\)’ refers the maximum number of clones of ‘\(a\textsubscript{v}\)’ and ‘\(K\textsubscript{v}\)’ refers the total number of agents available in the particular testing environment ‘\(v\)’.

Since this multi-agent framework provides, scalar type testing environment, at any instant, \(\text{Agents}(P\textsubscript{1}) \cap \text{Agents}(P\textsubscript{2}) = \emptyset\), where, \(P\textsubscript{1}\) and \(P\textsubscript{2}\) are different products (applications to be tested). i.e. at any specific service duration, neither a single agent (distributor) nor an agent set (testing agent + clones) can be shared by more than one product simultaneously.

The logical layout of the proposed MAS is shown in the Figure 6.1. As described earlier, there are three sets of components as distributor agents \(\{D_1, D_2, \ldots, D_x\}\), testing agents \(\{a_1, a_2, \ldots, a_x\}\) and the clones of testing agent \(\{ac_{11}, ac_{12}, \ldots, ac_{1k-1}\} \ldots \{ac_{x1}, ac_{x2}, \ldots ac_{xK_{x-1}}\}\) each of which runs locally on different machines in a network. The links between the agents refers to the service level dependency among the agents of MAS. Such a dependency can represent the fact that one agent depends on another for a goal to be fulfilled, task to be performed, or a shared resource to be made available.
Figure 6.1 Logical view of the proposed MAS
The distributors and testing agents are responsible for supervision and coordination of the core activities of respective lower layer entities of the overall environment. After getting the assignment from the user, the distributor agent \(D_i\) will choose the primary testing agents and can define the set \(\{a_1, a_2, \ldots, a_x\}\). The process of defining the set of required agents depends on the testing techniques required for any product ‘\(P_i\)’. Then, based on the load, each primary testing agent will select one or more additional agents of its respective groups and define the sets of cloning agents. These additional agents are to be referred to as secondary agents or clones of primary testing agents.

Each testing agent is responsible for assigning tasks to itself, originating in the local machine (where the testing agent resides) and to its clones. The agents provide a specific output standard to which the task output must conform. The output standard depends on the task type and the required output properties such as number of test cases, defects found, time spent for testing, etc.

### 6.3 CONSTRUCTION OF DISTRIBUTOR AGENT

The key responsibility of the distributor agent in this system is to segregate and distribute the assignments (from the external world) to the required service (testing) agents. i.e. system co-ordination. This task involves three subtasks for which three layers are distinguished; define the required set of service agents and distribute the assignments to the defined set of service agents (communication layer), product analyses and selection (reactive layer) and collect the individual outputs of service agents and integrate them (deliberative layer).
The generic structure of the distributor agent in relation to its environment is shown in Figure 6.2. The input to the distributor agent ‘\(D_i\)’ is (IF\(_1\)) from the tester and it includes the set of the testing product, time specification for testing, defect detector estimations and the specification about the required testing techniques. This is transferred to the database and the reactive layer of ‘\(D_i\)’ assesses it. The output of the reactive layer is the estimated values of products’ complexity arranged in the non-increasing order (IF\(_2\)).

The high priority product will be considered for service at first and the specifications will be transferred to communication layer (IF\(_3\)) and the other products will be given (IF\(_2\) and IF\(_5\)) to the next distributor agent ‘\(D_j\)’ (if any in the organization). Based on the testing service specifications, the appropriate set of agents can be defined and identified through the negotiation process in the communication layer (IF\(_7\), IF\(_8\)). Then the assignments will be distributed to the identified set of testing agents (IF\(_{12}\)) and their outputs (IF\(_{21}\)) can be obtained for integration. The Environmental Test Reports from the identified testing agents (IF\(_{21}\)) will be integrated in the deliberative layer and then passed to the external world (IF\(_{22}\)). The distributor agent is responsible for all types of co-ordination activities in this system and the key concepts (reasoning abilities) required for all types of co-operation and co-ordination activities in association with the responsibilities of the distributor agent are described as follows:

*Reasoning about other agents:* It is a basic reasoning ability of any agent that the communication layer is capable to perform. An agent is able to reason about any specific situation and other agents’ processes, which refers to the knowledge on other agents, about their capabilities, their strategic preferences, their assumptions, etc. This type of reasoning is essential for negotiation processes particularly in MAS.
Figure 6.2 Structure of Distributor Agent

\[ \text{IF}_1 = \{P_1, P_2, \ldots, P_n\} \text{ and their specifications} \]

\[ \text{IF}_4 = \{P_1, P_2, \ldots, P_n\} - \{P_i\} \text{ and their specifications} \]

\[ \text{IF}_{12} = \{P_i, A_i, T_{p_i}, \text{Defect Detector Information of } P_i, \text{ and Specification on } P_i\} \]

\[ \text{IF}_3 = \{P_i\} \text{ and its specifications} \]

\[ \text{IF}_{21} = \text{Environmental Test Reports from individual testing agents} \]

\[ \text{IF}_{22} = \text{Integrated Test Report} \]

---

Information Flow outside of the Agent

Information Flow within the Agent

Control Flow inside and outside of the Agent
Reasoning about interaction between agents: To interact with other agents, the agents must be capable of reasoning about interaction between agents. Agents are not only able to reason about which information can be sent or obtained by communication, but also know about how and when the communication has to be initiated. In the Figure 6.2, the set of information flows \{IF_2, IF_7, IF_8, IF_{10}, IF_{11}\} and the set of control flows \{CF_4, CF_5, CF_6\} are the examples for these two distinct reasoning in our system.

Reasoning about agent’s specific tasks: It includes the reasoning about the tasks that an agent or a part of it is to perform; about the way in which, a task is to be approached. This reasoning is generic in the sense that all autonomous agents are assumed to be capable of performing. This reasoning is implemented in our system through the sets of information and control flows \{IF_1, IF_2, IF_3, IF_6, IF_{21}\} and \{CF_2, CF_3, CF_7, CF_8\} respectively as shown in the Figure 6.2.

Reasoning about interaction with external world: In general, as autonomous agents are capable of interacting with the external world, the distributor agents must also be capable of reasoning about interaction with the external world; for example, the types of information that can be observed from the external world and when and how they can be observed. The sets of information and control flows \{IF_1, IF_{22}\} and \{CF_1\} as in the Figure 6.2 are the examples for representing the necessity of this particular reasoning in our system.

6.3.1 Communication Layer

The communication layer supports resource negotiation and allocation in the developed system. This layer includes the subtasks of testing techniques classification, agent registration and task distribution. Testing technique
classification module is responsible for identifying and classifying the required testing techniques. The agent registration module takes the responsibility of negotiation by keeps track of registered products with respect to the types of testing services and reply deadline. The task distribution module defines the scheduling of subtasks necessary to fulfill the testing requests and monitoring the services provided by the testing agents.

The distributor agents’ knowledge about the required testing techniques is crucial to the agents’ ability (i.e. system coordination activities). In the individualist agent environments, agents are viewed as independent entities; whereas in the collectivist environments, agents are interdependent. The presence of interdependent entities enables the cooperation and coordination activities. In this work, the proposed MAS is a collectivistic environment and the various forms of interdependencies in the system ‘S’ are as follows:

- The distributor agents as in the set \( \{D_1, D_2, \ldots, D_z\} \) depend on the testing agents in the set \( \{a_1, a_2, \ldots, a_x\} \) for the completion of the testing assignments.
- The set \( \{a_1, a_2, \ldots, a_x\} \) depends on the set \( \{D_1, D_2, \ldots, D_z\} \) for assignments.
- The set \( \{a_1, a_2, \ldots, a_x\} \) depends on the cloning agents in the sets \( \{ac_{11}, ac_{12}, \ldots, ac_{1K_{1-1}}\} \ldots \{ac_{x1}, ac_{x2}, \ldots, ac_{xK_{x-1}}\} \) respectively for the completion of the subtasks.
- The sets \( \{ac_{11}, ac_{12}, \ldots, ac_{1K_{1-1}}\} \ldots \{ac_{x1}, ac_{x2}, \ldots, ac_{xK_{x-1}}\} \) depend on the testing agents in the set \( \{a_1, a_2, \ldots, a_x\} \) for sub assignments.

Each agent belonging to an instance of the agent system S can communicate within its name space according to its behavior at any moment.
From the coordination point of view, the behavior of an agent determines the task distribution to other testing agents. For example, the set \(\{D_1, D_2, ..., D_z\}\) is the task distributor set for \(\{a_1, a_2, ..., a_x\}\) and the same \(\{a_1, a_2, ..., a_x\}\) can act as the distributors for their respective clones in the sets \(\{ac_{11}, ac_{12}, ..., ac_{1K_1-1}\} \cdots \{ac_{x1}, ac_{x2}, ..., ac_{xK_x-1}\}\).

On the other hand, from the cooperation point of view, agents are capable of doing the assigned tasks on their own. For example, the sets \(\{a_1, a_2, ..., a_x\}\), \(\{ac_{11}, ac_{12}, ..., ac_{1K_1-1}\} \cdots \{ac_{x1}, ac_{x2}, ..., ac_{xK_x-1}\}\) are capable of doing their assignments by their own. The individual agents have to be controlled according to the dependencies of the tasks assigned to them. Testing agents can communicate and co-operate with their distributor to achieve their instructions and goals. But the testing agents will not communicate with each other. i.e. there is no communication link between \(a_{it1}\) and \(a_{ij2}\) of the product ‘\(P_i\)’ such that, \(0 < j_1, j_2 \leq y\) and \(j_1 \neq j_2\). Similarly, the clones of any particular testing agent will have the links with their respective testing (parent) agent and need not to communicate with each other.

Communication between the software agents describes the ability to react in accordance to the incoming messages in exchange with other agents (Nwana et al 1996). i.e., communication, however, requires that the information transmitted by one agent results in a state change of another. Proving that the communication has occurred, however, requires us to know that the inner state of the receiving agent has in fact changed. Interaction, not only defines exchange of information, but also confirms that the other agent in fact received the original transmission. In other words, the original agent can infer that its transmission was communicated to other agent as soon as a
response is received - even if the response communicates that the responder did not understand the original message.

There are various forms of interactions such as unilateral, bilateral and multilateral models are used in the system. In this communication layer, the notion of workflow describes the complex coordination interactions in terms of client-server conversations. As shown in the Figure 6.2, the agent registration scheme uses three-stage conversation (bilateral interaction) model whereas the task distribution scheme uses four-stage conversation (multilateral interaction) model.

Conversation Schemes in Agent Registration:

- **request stage**: The distributor ‘Di’ send the requests to the identified testing agents of the product ‘Pi’ in the set \{a_{i1}, a_{i2}, \ldots, a_{ix}\}.
- **accept / reject stage**: The testing agents in the set \{a_{i1}, a_{i2}, \ldots, a_{ix}\} will send the response to the requests originator ‘Di’. The response may be positive (accepted the offer) or negative (rejected the offer) based on the status of the testing agents.
- **registration stage**: If the response from ‘av’ of the set \{a_{i1}, a_{i2}, \ldots, a_{ix}\} is positive, then ‘av’ will be added as ‘aij’ as a member in the set \{a_{i1}, a_{i2}, \ldots, a_{iy}\} of the product ‘Pi’.

Conversation Schemes in Task Distribution:

- **offer stage**: The distributor ‘Di’ offer the assignments to the agents in the set \{a_{i1}, a_{i2}, \ldots, a_{iy}\} of the product ‘Pi’.
- **confirm stage**: The testing agents in the set \{a_{i1}, a_{i2}, \ldots, a_{iy}\} will send the acknowledgements to the offers to the
originator ‘Di’. The response must be positive on acceptance of the assignment.

- **execution stage:** The testing agents in the set \( \{a_{i1}, a_{i2}, \cdots, a_{iy}\} \) execute the assignments offered based on the given optional specifications and will return the ETRs to the respective distributor ‘Di’.

- **termination stage:** After sending the ETRs to the respective distributor (Di) of the product ‘Pi’, the agents in the set \( \{a_{i1}, a_{i2}, \cdots, a_{iy}\} \) will terminate the logical link with their ‘Di’ and will become the member of the set \( \{a_1, a_2, \cdots, a_x\} \).

Coordination is necessary to keep track of dependencies between the distributed, heterogeneous, and autonomous subsystems (Lewis et al 1993). As different subsystems are involved, the dependencies may be arbitrarily complex. A fundamental aspect of interoperability is to keep track of such dependencies and reestablish overall consistency whenever necessary. Here the aim of co-ordination is described by the distributed and decentralized agreement of the behavior of the sets \( \{a_1, a_2, \cdots, a_x\} \) and \( \{ac_{11}, ac_{12}, \cdots, ac_{1K_1-1}\} \cdots \{ac_{x1}, ac_{x2}, \cdots, ac_{xK_x-1}\} \).

Coordination makes it possible that the task execution taken over by a group of agents takes place in a coherent, controlled process (Garrido et al 1996, Nwana et al 1997, Heiko et al 1999). To this end, the distributor agent has to compile and manage an agenda. The structure of the agenda contains entries about the specialized agents (technique based testing agents) involved, the task taken over, and about the result. The distributor agent recognizes the structure of the task and decomposes it down into sub-tasks. When an agreement has been reached, a contract is made and the sub-
task is handed over to the specified agent negotiating about the degree of control.

This communication layer provides the coordination processes rely on the underlying subsystems to provide key transactional functionality, which is not always present since we deal not only with database management systems but also with arbitrary applications. Distributor agent will provide this functionality on top of the subsystems. i.e. the transactional flows of the distributor agent are also concerned with other agents as the sets of information and control flows \{IF_1, IF_5, IF_7, IF_8, IF_10, IF_11, IF_22\} and \{CF_1, CF_4, CF_5, CF_6\} as in the Figure 6.2.

The negotiation model implemented here is an integrated model that enables the agents to qualitatively manage their attitude towards each negotiation session and provides the features of both cooperative and competitive models. The competitive model is used for testing product selection, cooperative model is acquired in agent registration scheme and the forced cooperative model is designed for task distribution scheme. For the feasibility of hybrid negotiation models in this work, some of the assumptions are made as described as follows:

*Nature of Agents:* This multi-agent framework provides, scalar type testing environment, i.e. one agent can provide the testing environment with only one testing technique. In the set \{a_1, a_2, \ldots, a_x\}, ‘x’ is the number of testing agents and also the number of testing techniques available in the MAS. On the other hand, in the sets \{ac_{11}, ac_{12}, \ldots, ac_{1K_1-1}\} \ldots \{ac_{x1}, ac_{x2}, \ldots ac_{xK_x-1}\}, ‘K_1-1’ is the number of clones or duplicates of the testing agent ‘a_1’ and similarly, ‘K_x-1’ is the number of clones or duplicates of the testing agent ‘a_x’.
Nature of Jobs: The description of the agents’ behavior in the previous sections implies that assignments are one-shot affairs; that is, they have definitely finite duration. This is well suited for describing jobs like testing time estimation, load sharing, etc. But it doesn’t fit more open-ended tasks such as defining the sets \( \{ D_i, a_{i1}, a_{i2}, \ldots, a_{iy} \} \) and \( \{ ac_{i(j1)}, ac_{i(j2)}, \ldots, ac_{i(k-j \cdot 1)} \} \) since these definition processes are based on the availability of the agents as specified in above sets.

Job priorities: Jobs are assigned priorities by the distributor agent and this is determined by the overall complexity of the products. Since we are trying to concentrate towards the complexity issues, we use the heuristic of giving highest priority to the products with highest estimated complexity.

Successful Service: For successful service, the set of agents allocated for any product must be less than or equal to the set of agents that are idle (the agents are not participated in any of the current service). i.e. for any ‘\( P_i \)’, \( A_{P_i} \leq B \), where ‘B’ is the set of idle agents in the system.

Mutual Exclusion: At any instant, \( Agents(P_1) \cap Agents(P_2) = \emptyset \), where, \( P_1 \) and \( P_2 \) are different products (applications to be tested). i.e. at any specific service duration, neither a single agent (distributor) nor an agent set (testing agent + clones) can be shared by more than one product simultaneously.

In the testing product selection, the preference will be assigned to the products such that \( \rho(P_1) \geq \rho(P_2) \geq \rho(P_3) \geq \cdots \geq \rho(P_n) \) such that, \( C(P_1) \geq C(P_2) \geq C(P_3) \geq \cdots \geq C(P_n) \) where, \( C(P_i) \) refers to the overall complexity of the product to be tested. Once the products are arranged in the above said order, the corresponding request will be processed one by one in accordance to the preference given to them.
In this collectivistic environment, although the distributor agents \( \{D_1, D_2, \ldots, D_z\} \) act as the coordination entities in the system, they have no inherent control over other agents in the sets \( \{a_1, a_2, \ldots, a_x\} \) and \( \{ac_{11}, ac_{12}, \ldots, ac_{1K_1-1}\} \ldots [ac_{x1}, ac_{x2}, \ldots ac_{xK_x-1}] \). So the only way they can influence the testing agents’ behavior is by persuasion. The forced co-operation is designed in the task allocation scheme. In this, the co-operation can be obtained through standardized sequence of actions and the negotiation would not be required. This is due the cooperation of other agents have been guaranteed in the registration process prior to this task allocation. In either case, the minimum requirement for negotiation is the distributor agents to be able to make proposals to the testing agents.

6.3.2 Reactive Layer

The prime responsibility of the reactive layer is to select a particular product for testing. Here, an automated negotiation model is developed based on the complexity attributes of the software that is to be tested with respect to the system used for testing the software. Irrespective of the critics of complexity measures either based on simple syntactic measures or sophisticated semantic measures, here the option is left to the designer to employ any kind of complexity assessment scheme. Since the input for this layer is designed as the composition of products to be tested and the corresponding complexity values, the scheme of complexity estimation is to be done prior to define the inputs for the MAS developed in this research.

Based on the complexity values, the products, which are assigned for testing, will be ordered in a non-increasing order. Let \( C(P_i) \) be the complexity of the product \( P_i \) and \( \rho(P_i) \) be the preference given to the product \( P_i \). In the ordered list, the products are arranged such that, \( C(P_1) \geq C(P_2) \geq C(P_3) \geq \ldots \)
\[ \cdots \geq C(P_H), \text{ i.e. the product with high complexity value will be in the top of} \]
\[ \text{the list and the product with less complexity value will be in the bottom of the} \]
\[ \text{list. The preference will be given to the products such that} \rho(P_1) \geq \rho(P_2) \geq \rho(P_3) \geq \cdots \geq \rho(P_H). \text{ So, whenever the testing process begins, the products will} \]
\[ \text{be selected one-by-one from the list. i.e. the product with high complexity} \]
\[ \text{value (C(P_1)) will be selected at first and so on.} \]

6.3.3 Deliberative Layer

The deliberative layer takes the responsibility of integrating the individual Environmental Test Reports (ETRs) received from the set \( \{a_{i_1}, a_{i_2}, \cdots, a_{i_y}\} \) and the output of this layer is Integrated Test Report (ITR). Each agent will send an ETR to its respective distributor agent after the completion of task assigned to it. Once the deliberative layer processes an ETR, if the layer has decided that other ETR of the same product exists, the new ETR has to be incorporated into that ITR. The internal structure of this layer is very simple and its responsibility is only to consolidate the individual test reports from the independent testing agents.

The ITR consists of the group of ETRs and a consolidated report. The consolidated report includes the specification of the product and the output of individual testing agents. The product specification contains the size of the product in LOC, average function size of the product, estimated complexity of the product, total number of test cases built for that particular product ‘\( P_i \)’, total number of errors found during the entire testing process, etc. The output of the individual testing agents consists of number of test cases of any particular testing agent and the number of errors found by the same set of test cases. Sometimes it also includes the time of completion of the testing processes. These details will be used as the history information to predicate the number of test cases to be built for each agent or product in future. The
testing agents can do this prediction and the results will be applied for other products or another version of the same product.

6.4 CONSTRUCTION OF TESTING AGENT

The primary components of the testing agents include automated units for test driver generation, test case execution, test case prediction and learning. The other activities like test data generation, test case design, test report generation have to be proceeded with manual assistance. The abstract view of the testing agent is shown in the Figure 6.3. The overall functionality of the testing agent is composed in the three layers: automatic driver generation and test case execution are implemented in the reactive layer; test case and test time prediction are to be done in the deliberative layer; the process of cloning and task distribution are implemented in the communication layer.

The links between the layers and the other entities represents the information and control flows between them. The initial input (IF\textsubscript{10}) to the testing agent a\textsubscript{ij} is from the corresponding distributor agent. It includes the set of the testing product, time specification for testing, defect detector estimations and the estimated complexity of the product. This is transferred to the database and the deliberative layer of a\textsubscript{ij} assesses it.

The output of the deliberative layer (IF\textsubscript{12}) is a set of estimated values on average size of the modules of the product and the predicted values of total number of test cases to be built by a\textsubscript{ij}, average size of the test case and average time required for generating and executing an unit test case.
Figure 6.3 Structure of Testing Agent
Communication layer will define the mode of load distribution and it is based on the input. This layer is also responsible for defining the number of clone(s) that are needed to generate. Moreover processes of clone registration (IF\textsubscript{14}, IF\textsubscript{15}), load distribution and collection of results from the clones (IF\textsubscript{16}, IF\textsubscript{17}, IF\textsubscript{18}, IF\textsubscript{19}) are to be done in the communication layer. At the same time it will distribute the load (IF\textsubscript{13}) to the reactive layer of same a\textsubscript{ij}. The results from the reactive layer (IF\textsubscript{20}) and Environmental Partial Test Reports (EPTRs) from the clones (IF\textsubscript{19}) will be processed in the deliberative layer. Then the generated Environmental Test Report (ETR) of a\textsubscript{ij} will be transferred to the tester and the distributor agent D\textsubscript{i} (IF\textsubscript{21}).

From the perspective of types of reasoning, the sets of information flows \{IF\textsubscript{14}, IF\textsubscript{15}, IF\textsubscript{16}, IF\textsubscript{17}, IF\textsubscript{21} \} and control flows \{CF\textsubscript{5}, CF\textsubscript{8}, CF\textsubscript{9} \} in the Figure 6.3, are the examples for reasoning about other agents and reasoning about interaction between agents in the testing agent. Then the set of information flows \{IF\textsubscript{8}, IF\textsubscript{9}, IF\textsubscript{10}, IF\textsubscript{11}, IF\textsubscript{12}, IF\textsubscript{13}, IF\textsubscript{18}, IF\textsubscript{19}, IF\textsubscript{20} \} and the set of control flows \{CF\textsubscript{7}, CF\textsubscript{10}, CF\textsubscript{11} \} are the examples for reasoning about agents’ specific task in the testing agent.

6.4.1 Communication Layer

The communication layer supports cloning and task distribution among the testing agents in the proposed system. The arrangement of primary components of the communication layer is shown in the Figure 6.3. This layer provides three modes of task distribution and two modes of cloning. Three modes of task distribution include 100% load to the resident agent (Mode 1), 20%-80% load distribution (Mode 2) and multiple agents load distribution (Mode 3). In the first scheme, the reactive layer of the resident agent a\textsubscript{ij} can handle the full load. In this scheme, load-sharing and cloning are not necessary. In the other two schemes, cloning should be committed for load
sharing. The clone registration module is responsible for negotiation by keeps track of the agents registered for service and with the reply deadlines. The task distribution module takes the responsibility of scheduling of load distribution necessary to fulfill the testing requests and monitoring the services provided by the clones.

It is explained that the presence of interdependent entities enables the cooperation and coordination activities and various forms of interdependencies are implemented in the system ‘S’ with respect to the distributor agent. From the coordination point of view, the behavior of a testing agent determines task distribution to its clones. For example, the set \( \{a_{i1}, a_{i2}, \ldots, a_{iy}\} \) can act as the distributors for their respective clones sets\( \{ac_{i(11)}, ac_{i(12)}, \ldots, ac_{i(1K_{i1}-1)}\} \ldots\{ac_{i(y1)}, ac_{i(y2)}, \ldots, ac_{i(yK_{iy}-1)}\} \).

From the cooperation point of view, testing agents are capable of doing the assigned tasks on their own. For example, the sets \( \{a_{i1}, a_{i2}, \ldots, a_{iy}\} \), \( \{ac_{i(11)}, ac_{i(12)}, \ldots, ac_{i(1K_{i1}-1)}\} \ldots\{ac_{i(y1)}, ac_{i(y2)}, \ldots, ac_{i(yK_{iy}-1)}\} \) are capable of doing their assignments by their own. The individual agents have to be controlled according to the dependencies of the tasks assigned to them. As shown in the Figure 6.3, three-stage conversation (bilateral interaction) model is implemented in the clone registration scheme and four-stage conversation (multilateral interaction) model is used for the task distribution scheme for clones.

Conversation Schemes in Clone Registration:

1. Request stage: The testing agents in the set \( \{a_{i1}, a_{i2}, \ldots, a_{iy}\} \) send the requests to the respective identified clone(s) in the set \( \{ac_{i(11)}, ac_{i(12)}, \ldots, ac_{i(1K_{i1}-1)}\} \ldots\{ac_{i(y1)}, ac_{i(y2)}, \ldots, ac_{i(yK_{iy}-1)}\} \).
2. Accept / reject stage: The clone(s) in the set 
\( \{ac_{i(1)}, ac_{i(2)}, \ldots, ac_{i(1K_i)} \} \ldots \{ac_{i(y1)}, ac_{i(y2)}, \ldots, ac_{i(yK_y)} \} \)
will send the response to the requests originator in the set 
\( \{a_{i1}, a_{i2}, \ldots, a_{iy} \} \). The response may be positive (accepted the offer) or negative (rejected the offer) based on the status of the clones.

3. Registration stage: If the response from the clone(s) of the set 
\( \{ac_{i(1)}, ac_{i(2)}, \ldots, ac_{i(1K_i)} \} \ldots \{ac_{i(y1)}, ac_{i(y2)}, \ldots, ac_{i(yK_y)} \} \) is positive, then the particular clone will be added as a member of ‘\( a_{ij} \)’ as in the set \( \{a_{ij}, ac_{i(j1)}, ac_{i(j2)}, \ldots, ac_{i(jK_i)} \} \) of the product ‘\( P_i \)’.

Conversation Schemes in Task Distribution:

1. offer stage: The testing agent ‘\( a_{ij} \)’ offer the assignments to its clones in the set \( \{a_{ij}, ac_{i(j1)}, ac_{i(j2)}, \ldots, ac_{i(jK_i)} \} \) of the product ‘\( P_i \)’.

2. confirm stage: The clones in the set \( \{a_{ij}, ac_{i(j1)}, ac_{i(j2)}, \ldots, ac_{i(jK_i)} \} \) will send the acknowledgements to the offers to the originator ‘\( a_{ij} \)’. The response must be positive on acceptance of the assignment.

3. execution stage: The cloning agents in the set \( \{a_{ij}, ac_{i(j1)}, ac_{i(j2)}, \ldots, ac_{i(jK_i)} \} \) execute the assignments offered based on the given optional specifications and will return the EPTRs to the respective testing agent ‘\( a_{ij} \)’.

4. termination stage: After sending the EPTRs to ‘\( a_{ij} \)’, the clones in the set \( \{a_{ij}, ac_{i(j1)}, ac_{i(j2)}, \ldots, ac_{i(jK_i)} \} \) will terminate the logical link with their corresponding testing agents.
The set of testing agents \( \{a_{i1}, a_{i2}, \ldots, a_{iy}\} \) are capable of generating clones to assist for task completion. i.e. the load will be shared by multiple identical testing agents of same group. In this mode, the testing agent \( a_{ij} \) can be defined as, \( a_{ij} = \{a_{ij}, ac_{i(j1)}, ac_{i(j2)}, \ldots, ac_{i(jK_{ij} - 1)}\} \) where, ‘\( K_{ij} \)’ indicates the total number of agents (testing agent and its clones) and ‘\( K_{ij} - 1 \)’ denotes the number of clones of the testing agent \( a_{ij} \) in the specific testing environment.

For example, the set of testing agents \( \{a_{ij}, ac_{i(j1)}, ac_{i(j2)}, \ldots, ac_{i(jK_{ij} - 1)}\} \) are to be referred as the clones of the testing agent ‘\( a_{i1} \)’. The agent \( a_{i1} \) can get the assignment from its distributor \( D_i \) and \( a_{i1} \) distribute the assignments to its clones \( \{a_{ij}, ac_{i(j1)}, ac_{i(j2)}, \ldots, ac_{i(jK_{ij} - 1)}\} \).

The internal structure of the cloning agents is identical to the respective testing agent of the same group in that they have exactly same behavior. But once a particular agent is being registered as a clone, the responsibility for the clone is restricted such that they are nominated for testing purpose only and no need to operate in the distribution mode. Cloning agents can communicate and co-operate with their originator (corresponding testing agent) to achieve their instructions and goals. But they would not communicate with each other. Each principle partner (clone) can manage and control itself on a local dimension and interact directly with their originator to exchange, provide and receive services, data and knowledge. There are two schemes available for cloning process: 20% - 80% classification scheme and Time-based cloning.

20% - 80% classification scheme: In this scheme, pareto principle is applied for load sharing. Here only one additional agent will be added with testing agent \( a_{ij} \), i.e. only one clone will be generated such that \( a_{ij} = \{a_{ij}, ac_{i(j1)}\} \) where, \( K_{ij} = 2 \). In the total load, 20% of the modules, in
which the defect detector value is high, will be given to the clone \( ac_{i(j_1)} \) and the remaining load (80% of the modules) will be processed by the testing agent \( a_{ij} \).

**Time-based cloning:** In this mode, total load is shared based on the time specification \((T_{P_i})\) supplied along with the product \( P_i \). For this, the communication layer must be aware of number of clones need to be generated. The number of clones can be estimated such that the time required by any agent in the set \( \{a_{ij}, ac_{i(j_1)}, ac_{i(j_2)}, \cdots, ac_{i(j_{K_{ij}}-1)}\} \) should be less than or equal to \( T_{P_i} \).

So, for any specific testing environment, \( K_{ij} - 1 \) indicates the number of clones needs to be generated such that, \( T_{P_i} \geq T_{a_{ij}} \) and \( T_{P_i} \geq T_{ac_{i(ju)}} \). Here, \( T_{a_{ij}} \) refers to the total time required for testing in the agent \( a_{ij} \) and \( T_{ac_{i(ju)}} \) refers to the total time required for agent \( ac_{i(ju)} \). The addition of clones for particular environment will reduce the total processing time but there will be an increment in number of testers in the manual testing. Of course, if one adds more agents (clones) in a particular environment, then the system throughput will always improve irrespective of the type of testing – either manual or automated.

### 6.4.2 Reactive Layer

The reactive layer supports test case generation and execution processes in the proposed system. This layer includes module selector, automatic driver generation and test case generation and execution. Module selector is responsible to scan the product \( P_i \) and display the set of all functions or modules in \( P_i \) along with an option to the tester to select the
module to be tested. An alternative way of module selection is based on the defect detector value. This helps to follow the pareto principle in which the preference will be given to the high complexity modules. Here we used the traditional method of generating detector, which uses the threshold value of Cyclomatic Complexity measures (McCabe 1776, Stephen 2000, Tim et al 2000, Stephen 2001, Tim et al 2003, Tim et al 2004).

Compared to the development process of a product, generating test drivers for the same is a difficult problem. In this research, initially the drivers are based on an interpreter model: a test driver can be viewed as a command interpreter that reads the test cases and translates them into actions on the component under test. From this point of view, it is straightforward to parse a component's interface definition, identify its operations, and construct an interpreter. The major weakness of this approach is ineffective way of handling the components that rely on inversion of automation principle or that have a substantial human interaction component. Hence to avoid these, a compiler model test driver has been implemented and described here. A test driver generator based on this strategy has been designed and is being implemented and the ‘C’ language serves as the underlying implementation language for components in this work.

The architecture for the test driver uses the envelope and letter paradigm for handling internal values. As a result, the driver can directly refers to the unit under test or any of its components. The execution module has to be tailored with the basic requirements of the underlying subsystem. That means, the support for any unit under test can be directly added without requiring any changes, rather than the driver generator creates a “glue” source file, which will be compiled and then linked with the existing object files, produces a custom driver for the component under test. Preliminary experiences with this approach indicate that it provides significant time
savings when large test sets are used. This is also true for all invocations of subsystem operations and offers the provision of atomicity by step-wise undoing the effects of failed activities.

The test case specifications are depends on the technique underlying in the system or system agents. In case test case generation, activities (test case specifications) specified by the tester have to be mapped to local operations. This includes the preservation of the given orders as well as the provision of local atomicity. As discussed in the experimentation methodology (Chapter 3), here three basic testing techniques of white box models are used. Based on the technique, the testing object or section can be selected by the tester.

### 6.4.3 Deliberative Layer

The prime responsibilities of the deliberative layer are testing time prediction and report generation. The set of constants \( \{Q_1, Q_1, \ldots, Q_7\} \) of the regression models are distinct for each product. The estimations of the constants are based on the history information on the test results of past products. For each product, testing will be started using the predicted values based on the history information. After the testing is completed, the actual records will replace the predicted values. i.e. initially the load distribution is based on the predicted values. So, the results on each service will be updated and thus the estimated values of \( \{Q_1, Q_1, \ldots, Q_7\} \) are distinct and not same for more than one product. This iterative nature of learning process enables the refinement of regression model constants for large number of samples.

The deliberative layer takes the responsibility of Environmental Test Report (ETR) generation. This might be based on the results received from the reactive layer or the combination of the results from the reactive layer of
as well as Environmental Partial Test Reports (EPTRs) from its clone(s). The testing report includes the preservation of the given orders as well as the provision of local atomicity for generating reports on execution modules. For these purposes, the testing agent uses a DBMS for storing the results together with the associated parameters. If the underlying subsystem is based on single database and if all the testing agents have access to this DBMS, it can be exploited; otherwise, a separate DBMS has to be made available for each testing agent.

6.5 SUMMARY

The MAS as a test bed constructed in such a way that it provides the significant merits of MAS at different levels, which allows the MAS designers to focus on the conceptual design, rather than on low-level system programming issues. In other words, here the focus of the MAS construction is completely oriented towards realizing a real time MAS environment in order to facilitate investigations in such a way to validate and justify the proposed metrics models, rather than assessing the effectiveness of the proposed MAS in enhancing the performance of the intended application domain. Hybrid architectures are used for building different types of agents in the proposed system. This made to combine the benefits of deliberative and reactive agent architectures. Three-layer approach is followed for building all types of agents in the constructed MAS.

In case of distributor agent, reactive layer takes the responsibility of selection of particular product for testing service, where the selection is based on the testing complexity values of the products. The resource negotiation and allocation schemes are implemented in the communication layer. Various forms of interdependencies and interaction models are described in this communication layer. A class of resource negotiation and allocation
algorithms are proposed and discussed. Deliberative layer is responsible for integrating the environmental test reports of the individual testing agents. In short, the distributor agent is responsible for overall system coordination that defines the scheduling of subtasks necessary to fulfill the testing requests.

The overall functionality of the testing agent is implemented in the three sub layers as in the distributed agents: deliberative layer, communication layer and reactive layer. The activities of the deliberative layer are defined as testing time prediction, automated learning and test report generation. In case of testing time prediction, varieties of regression models can be used for test case prediction and total time prediction. The constants of regression models are used for environment learning and updation. The necessity of cloning is explained along with the clone registration and task allocation algorithms in the communication layer. The modules for automatic driver generation, test case generation and execution schemes are implemented in the reactive layer. In this work, three basic testing techniques of white-box models are implemented. Test cases generated for experimentation purpose are based on the testing techniques and the corresponding coverage criteria.