International Journal of Science and Research (IJSR)
ISSN: 2319-7064
ResearchGate Impact Factor (2018): 0.28 | SJIF (2018): 7.426

Designing an Intelligent Software Agent with DECAF Multi-Agent Platform

Sanjay Yede[1], V. N. Chavan[2]

[1]P.G. Department of Computer Science and Technology, Degree College of Physical Education, HVPM, Amravati-444605, India

[2]Head, Department of Comp. Sci, Seth Kesarimal Porwal College, Kamtee, Nagpur, India

Abstract: ***DECAF (Distributed, Environment Centred Agent Framework) is a software platform that helps design, development, and execution of "intelligent" software agents[15], to achieve solutions in complex problem domains. It provides an environment for the execution of intelligent agents. The environment includes the ability to communicate with other agents, efficiently maintain the current state of an executing agent, and select an execution path from a set of possible ones, so as to support persistent, flexible, and robust actions. DECAF provides a modular platform for evaluating and disseminating results in agent architectures, including communication, planning, action scheduling, execution monitoring, coordination, and learning. By modularizing the design of the software, researchers can analyze and focus on specific issues in agent development, coordination and planning without disturbing other parts.***

Keywords: Multi-Agent System, MAS architecture, Agent Communication, DECAF components

1. Introduction

DECAF is an agent development and execution environment that allows a well-defined approach towards building multi-agent systems. It provides necessary services of a large-grained intelligent agent,[3,12] like: communication, planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis [5].

Agent development in a Multi-agent systems faces problem of constructing a robust environment for executing agent tasks. This environment must be able to use and understand network and communication protocols, adapt in the face of failure, and provide a platform for development of the agent tasks themselves. Then the tasks must be organized (programmed) to provide the "intelligence" of the agent code, and multi-agent activity must be supported and coordinated via scheduling and communication protocols [11]. DECAF helps in prototyping MAS by taking care of certain common details via reusable behaviors [2] and providing standardized, domain-independent or easily customizable middle agents [4]. Reusable behaviors include things such as Agent Name Server registration and deregistration, Agent Management protocols, and service negotiation via middle agents such as matchmakers. DECAF middle agents include agent name servers, matchmakers, brokers, information extraction agents, web proxies, and agent management agents.

2. DECAF Architecture

Following figure represents the high level structure of the DECAF architecture [1]. There are five internal execution modules and seven associated data structure queues in the current DECAF implementation.

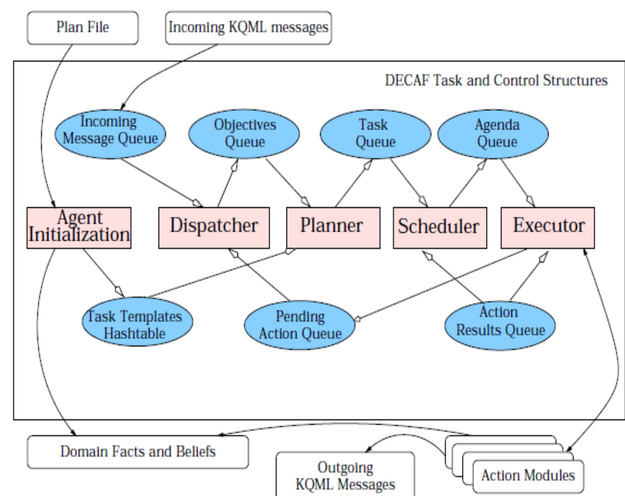

Figure 1: The DECAF architecture [7]

2.1 Agent Initialization

The execution module is responsible for controlling agent's task-flow in its entire life time. After initialization, each module runs continuously and concurrently in its own thread. An agent's execution starts with the execution of *Agent Initialization Module*. The *Agent Initialization* module reads the plan file(s). Each task reduction specified in the plan file is added to *Task Templates Hash table, also called as* plan library. Next, the plan may make use of a *Startup module*. The Startup task of an agent might, for example, build any domain database/knowledge- base needed for future execution of the agent, or register with a Matchmaker. Any subsequent changes to initial data must come from the agent actions during the completion of goals. Startup tasks may assert certain continuous maintenance goals or initial achievement goals for the agent. The Startup task is special since no message will be received to begin its execution. If such a Startup module is part of the plan file, the initialization module will add it to the *Task Queue* for immediate execution. Finally, the Agent Initialization Module does register with the ANS and set up all socket and network communication.

Licensed Under Creative Commons Attribution CC BY

2.2 Dispatcher

Agent initialization is done once and then control is passed to the *Dispatcher* which waits for incoming KQML messages which are placed on the *Incoming Message Queue*. An incoming message contains a KQML *performative* and its associated information becomes an objective indicating which capability within the agent is to be accomplished. An incoming message can result in one of three actions by the dispatcher,

- The message is attempting to communicate as part of an ongoing conversation. The Dispatcher makes this distinction mostly by recognizing the KQML in-reply-to field designator, which indicates the message is part of an existing conversation. In this case the dispatcher will find the corresponding action in the *Pending Action Queue* and set up the tasks to continue the agent action.
- The message indicates that it is part of a new conversation. This will be the case whenever the message does not use the, 'in-reply-to' field. If so a new *objective* is created and placed on the *Objectives Queue* for the Planner. The dispatcher assigns a unique identifier to this message which is used to distinguish all messages that are part of the new conversation.
- The dispatcher is responsible for the handling of error messages. If an incoming message is improperly formatted or if another internal module needs to sends an error message the Dispatcher is responsible for formatting and sending the message.

2.3 Planner

The Objectives Queue at any given moment will contain the instantiated plans/task structures (including all actions and sub-goals) that should be completed in response to all incoming requests. The initial, top-level objectives are roughly equivalent to the BDI "desires" concept[14], while the expansion into plans is only part of the traditional notion of BDI "intentions", which for DECAF is divided into three reasoning levels, planning, scheduling, and execution. The *Plan scheduler* goes to sleep state when the *Objectives Queue* becomes empty and remains in sleeps state till the queue is empty. The purpose of the Plan Scheduler is to determine which actions are to be executed when and in what order[13]. This determination is currently based on whether all of the provisions for a particular module are available. Some provisions come from the incoming message and some provisions come as a result of other actions being completed. This means the objectives queue is checked any time a provision becomes available to see which actions can be executed now. The Planner monitors the Objectives Queue and matches new goals to an existing task template as stored in the Plan Library. A copy of the instantiated plan, in the form of an HTN corresponding to that goal is placed in the *Task Queue* area, along with a unique identifier and any provisions that were passed to the agent via the incoming message. If a subsequent message comes in requesting the same goal, then another instantiation of the same plan template will be placed in the task queue with a new unique identifier. The Task Queue at any given moment will contain the instantiated plans/task structures that should be completed in response to an incoming request.

2.4 Scheduler

The *Scheduler* waits until the Task Queue is non-empty. The scheduling functions are actually divided into two separate modules; the *Task Scheduler* and the *Agenda Manager*. The purpose of the Task Scheduler is to evaluate the HTN task structure to determine a set of actions which will "best" suit the users' goals. The input is a task HTN will reflect all possible actions, and the output is a task HTN pruned to reflect the desired set of actions. Once the set of actions have been determined, the Agenda Manager (AM) is responsible for setting the actions into execution. This determination is based on whether all of the provisions for a particular module are available. Some provisions come from the incoming message and some provisions come as a result of other actions being completed. This means the Tasks Queue Structures are checked any time a provision becomes available to see which actions can be executed now. The other responsibility of the AM is to reschedule actions when a new task is requested. Every task has a window of time that is used for execution. If subsequent tasks can be completed while currently scheduled tasks are running then a commitment is made to running the task on time. Otherwise the AM will respond with an error message to the requester that the task cannot be completed in desired time.

2.5 Executor

The *Executor* is set into operation when the Agenda Queue is non-empty. Once an action is placed on the queue the Executor immediately places the task into execution. One of two things can occur at this point: The action can complete normally. (Note that "normal" completion may be returning an error or any other outcome) and the result is placed on the *Action Result Queue*. The framework distributes the result as provisions to downstream actions that may be waiting in the Task Queue. Once this is accomplished the Executor examines the Agenda queue to see if there is further work to be done. The other case is when the action partially completes and returns with an indication that further actions will take place later. This is a typical result when an action sends a message to another agent, requesting information, but could also happen for other blocking reasons (i.e. user or Internet I/O). The remainder of the task will be completed when the resulting KQML message is returned. To indicate that this task will complete later it is placed on the *Pending Action Queue*. Actions on this queue are keyed with a *reply-to* field in the outgoing KQML message[17]. When an incoming message arrives, the Dispatcher will check to see if an *in-reply-to* field exists. If so, the Dispatcher will check the Pending action queue for a corresponding message. If one exists, that action will be returned to the Agenda queue for completion. If no such action exists on the Pending action queue, an error message is returned to the sender.

3. Features of DECAF

DECAF provides an end-to-end environment for the execution of agent tasks. It provides an operating environment, where jobs are entered and executed to completion without user intervention. Also a set of system services for ease of software development have been

Licensed Under Creative Commons Attribution CC BY

provided. In order to support the development of agents, following tools are supplied with DECAF:

- **Plan Editor:** It is a GUI interface that allows graphic programming of agent tasks and actions. The Plan Editor allows editing of features needed to reason about scheduling activities, representation of message sending, use of library plans, and control flow logic.
- **Middle Agents:** These have been developed to support common multi-agent activities. A middle agent is an agent that facilitates agent operation while not directly related to completing a specific task.
- **Matchmaker:** The Matchmaker agent serves as a "yellow page service" to assist agents in finding services usual for task completion.
- **Broker:** A Broker agent acts as a "white pages" directory to assist agent with collections of services.
- **Proxy:** A proxy agent allows web page Java applets to communicate with DECAF agents that are not located on the same server as the applet.
- **Agent Management Agent (AMA):** AMA allows MAS designers to look at the entire running set of agents spread out across the internet that share a single agent name server. This allows designers to query the status of individual agents and watch message passing traffic.
- **Agent Name Server (ANS):** The ANS is an essential component for agent communication. It works in a fashion similar to DNS server by resolving agent names to host and port addresses.

4. The DECAF Functions

The DECAF agent architecture implements following five basic functions[16]:

- *Initialization:* I(PF) = {TT}

where, the initialization function I takes a plan file PF as input and produces a *set* of task templates TT as output. The set of task templates is a definition of the *capabilities* of this particular agent.

- *Dispatching* -D(M) = O

where, the dispatcher function D takes a KQML message M as input and produces a new objective O as output. The message M can be a new request or a response to an old request

- *Planning* - P(O,{TT}) = TQ

where, the planning function P takes an objective O and a set of task templates for that objective {TT} as input and produces an instantiation of the appropriate task template, known internally as a task queue object, TQ as output.

- *Scheduling* - S({TQ}) = {A}

where, the scheduling function S takes a set of task queue objects TQ as input and produces an agenda A as output. The agenda is a set of actions to perform. Simultaneously, S notes when actions have been completed and more actions can be enabled as a result.

- *Execution* - E({A}) = {IN}

where, the execution function E takes a set of enabled actions A as input and produces a set of "intentions" {IN} as output.

In summary, the set of actions to be executed by DECAF is denoted by the equation, {IN}=E(S(P(D(M),I({PF}))))

In colloquial terms, an instantiation of the architecture takes a plan file PF, and waits for a message, M. When the message arrives, there are three possibilities:

- The message specifies a task that is not in the capabilities of the plan file. In this case, the action taken is to reply the sender with an error message.
- The message specifies the start of a new task not previously requested. In this case, the architecture will follow each of the steps above and produce a set of actions to complete the task.
- The message is in response to a message sent to assist in completion of an ongoing task. In this case, the dispatching, planning and scheduling functions are skipped and the content of message is reported to the awaiting task.

5. The DECAF Components

The basic operation of DECAF requires three components: an agent name server (ANS), an agent program (plan file), and the agent framework of DECAF itself. The purpose of the ANS is similar to the most name servers such as DNS (Domain Name Server) which is used to bind an agent instance with a name or helps searching an agent instance based on given name. The agent's interactions with the ANS are transparent to the user and occur automatically. Currently DECAF uses the RETSINA [19] ANS protocol. The plan file is the output of the Plan Editor and represents the programming of the agent. One agent consists of a set of capabilities and a collection of actions that are planned and executed to achieve the objectives. The basic actions are implemented in the form of a computer program in C, java, etc. However, the expression of a plan for providing a complete capability is achieved via program building blocks that are not Java statements but are a sequence of actions connected in a manner that will achieve the goal.

Actions are reusable in any sequence for the achievement of many goals. Plan annotations can include alternative subgoals. In the Plan-Editor, a capability is developed using a HTN tree structure in which the root node expresses the entry point of this capability "program" and the goal to be achieved. Non-leaf nodes (other than the root) represent intermediate goals or compound tasks that must be achieved before the overall goal is complete. Leaf nodes represent actions. Each task node (root, non-root and leaves) has a set of zero or more inputs called *provisions*, and a set of zero or more *outcomes* [8]. The provisions to a node may either be forwarded from the provisions of a parent task, or come from the outcomes of different actions. No action will start until all of its provisions have been supplied by an outcome being forwarded from another node (this may be an external node representing some non-local task and the reception of a KQML or FIPA message[17,18]). Provision arcs between nodes represent the most common type of inter-task on the plan specification. A node may have multiple outcomes and is considered complete as soon as one outcome has been provided. The *outcomes* represent a complete *classification* or partition of the possible results. By connecting different node outcomes to different downstream input provisions in the Plan-Editor, conventional looping or conditional selection can be created.

Licensed Under Creative Commons Attribution CC BY

6. Conclusion

The DECAF Agent architecture is a general purpose agent development platform designed specifically to support concurrency, distributed operations, high level programming paradigms and high throughput. The architecture is highly threaded to adapt itself to multi-processor architectures**.** DECAF is based on the premise that execution of the actions required to accomplish a task specified by an agent program is similar to a traditional operating system executing a sequence of user requests. It supports programming at the multi-agent level via the use of several fairly standard prebuilt middle-agents and generally useful agent classes such as a web information extraction agent. The architecture makes much use of modern multi-threading and multiprocessor machines. It also provides support for persistent and flexible actions through the ability to program alternatives that are chosen dynamically at run-time, and plans with data-flow control based on multiple outcomes and input provisions allowing looping and other robustness-enhancing processes.

7. References

[1] Brooks R A, “A Robust Layered Control System for a Mobile Robot”, *Journal of Robotics and Automation IEEE,* 1986.
[2] Decker K S, Pannu A, Sycara K and Williamson, “Designing Behaviors for Information Agents”, In *Proceedings of the 1st Intl. Conf. on Autonomous Agents*, Marina del Rey, 1997a
[3] Decker K S and Sycara K, “Intelligent Adaptive Information Agents”, in *Journal of Intelligent Information Systems,* 1997.
[4] Decker, K S, Sycara K and Williamson M, “Middle-Agents for the Internet”, in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997b.
[5] Horling B, Lesser V, Vincent R, Bazzan A and Xuan P, “Diagnosis as an Integral Part of Multi-Agent Adaptability”, Tech Report CS-TR-99-03, UMass. 1999
[6] Graham J R and Decker K S, “Towards a Distributed, Environment-Centered Agent Framework”, in: N. Jennings and Y. Lesp´erance (eds.): *Intelligent Agents VI— Proceedings of the Sixth International Workshop on Agent Theories, Architectures,and Languages (ATAL-99*Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin. 2000
[7] Graham J R, Decker K S and Michael M, “DECAF - A Flexible Multi Agent System Architecture”, AAMAS-2001.
[8] Williamson M, Decker K S and Sycara K, “Unified Information and Control Flow in Hierarchical Task Networks”, in *Proceedings of the AAAI-96 workshop on Theories of Planning, Action, and Control*, 1996b
[9] Graham J R, “Real-time Scheduling in Multi-agent Systems”, PhD. Thesis, University of Delaware, 2001.
[10] Grosz B and Kraus S, “Collaborative Plans for Group Activities”, in *Proceedings of the Thirteenth International Joint Conference on A0rtificial Intelligence*, Chamb´ery, France. 1993
[11] Sycara K Decker K S, Pannu A, Williamson M and Zeng D, “Distributed Intelligent Agents”, *IEEE Expert* 1996.
[12] Harvey T and Decker K, “Planning Ahead to provide Scheduler Choice”, in *Proc. Of Workshop on Infrastructure for Scalable Multi-Agent Systems.* 5th International Conference on Autonomous Agents, 2001.
[13] Rao A and Georgeff M, “BDI Agents: From Theory to Practice”, in *Proceedings of the First International Conference on Multi-Agent Systems,* San Francisco, 1995.
[14] Wooldridge M and Jennings N, “Intelligent Agents: Theory and Practice”, *The Knowledge Engineering Review,* AAMAS, 1995.
[15] Cohen P R and Levesque H J, “Intention is Choice with Commitment”, *Artificial Intelligence*-1990.
[16] Tim Finin, Yannis L and Mayfield J, DRAFT:KQML as an agent communication language, Computer Science Department University of Maryland Baltimore,USA.
[17] FIPA. The Foundation for Intelligent Physical Agents. At http://www.fipa.org, 2001.
[18] Onn Shehory, “Architectural Properties of MultiAgent Systems”, The Robotics Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213, 1998

Licensed Under Creative Commons Attribution CC BY