An Assembly and Execution Shell for MultiAgent Systems

Glenn T. Jayaputera♣ Arkady Zaslavsky♣ Seng W. Loke♣ Nigel Watson♦
♣*{Glenn.Jayaputera, Arkady.Zaslavsky, Seng.Loke}@infotech.monash.edu.au*
Computer Science and Software Engineering, Monash University
900 Dandenong Road, Caulfield East, Victoria 3145, Australia
♦*nigelwat@Microsoft.com*
Microsoft Australia
9/644 Chapel Street, South Yarra, Victoria 3141, Australia

Abstract

This paper presents a novel approach to design and develop an assembly and execution shell for multiagent applications based on the concept of a mission. A mission is a live evolving object that contains data, current variables, goals and evolution history of the mission itself. By maintaining the states and data inside the mission, it is possible to suspend the multiagent system's mission execution temporarily and resume when permitted. Our design allows run-time and dynamic agent generation depending on the complexity of the mission itself. We discuss the design and internal structure of the mission, the strategy and efficiency of the system in executing the tasks in the mission, and present our prototype system called eHermes.

1 Introduction

Agent systems have been used in diverse applications. Each such system might have met their design requirements, but generally, they cannot be easily adapted to other (even related) problems. In most cases, the agent designer must create a new system perhaps from scratch to satisfy different requirements. Constructing multiagent systems (including the individual agents and the coordination mechanisms) in general is an arduous task and requires sophisticated software engineering methodologies and tools. We believe that, for certain classes of multiagent applications where the aim of the application (and therefore, the *overall purpose* or *mission* of the collection of agents for the application) can be explicitly identified and concisely expressed, it might be possible to automatically send that mission to a system for processing. Such system will generate agents on demand, at run-time and deploy plus coordinate those agents in order to complete the mission. Components can also be reused in generating agents for different missions.

Our vision is to have a system that has no knowledge about what to solve at the start, but when given a mission, it is able to work out a means to achieve the mission efficiently. This means that the number of agents generated and what functionality they have will depend on the complexity of the mission.

This paper presents our approach to developing an assembly and execution shell for multiagent systems based on the above-mentioned hypothesis. The shell will accept the best possible plan that a planner can provide, and execute the plan as a mission. We call our system a shell by analogy with expert systems shells in that it does not contain any expertise in itself, but expertise is supplied depending on the mission or problem to solve. In our approach, we employ a mechanism to assemble agent(s) on-demand and they are given tasks at run-time. The number of agents created depends on several factors, such as available system resources, cost and time. The user can control the system by specifying 'at the minimal cost' (hence only limited agents are created), 'at any cost' (hence multiple agents are running concurrently in order to get the results as soon as possible) or must be completed at a specific time (in this case the system will take a cost/efficiency analysis at all time. If for instance, a single agent is enough to do the mission within the time frame specified then the system would opt to do so). The system coordinates and manages those agents executing the mission as well maintains the history of execution of the missions. Missions can be suspended, resumed, aborted or modified at run-time. A prototype of our system, called eHermes, is being developed to realize our vision. eHermes differs from other toolkits that have an agent generation component such as Zeus [10], RETSINA [12] and DESIRE [2] to name a few, in that eHermes goes further, that is, it attempts to tightly manage the overall execution of the generated agents.

The rest of this paper is organized as follows. Section 2 presents the notion of the mission. We explain in detail our task decomposition model, strategies to execute the model safely and in parallel, the component that allow tasks to exchange their data and the component that maintains the history and data state of the task decomposition model. In Section 3 we describe our prototype system eHermes. We present in this section, eHermes conceptual diagram as an agent shell system, as well as its behaviour as soon as a request is supplied to it. Section 4 presents related work. Finally, we conclude with future work in Section 5.

2 The Mission Concept

The mission is the central idea of our work. We define a mission not only as the goal that a system must carry out rather as a "living" object that does not only encapsulate the objective of the mission but the whole evolution of that mission.

In our vision, a mission object should not only contain simple executable instructions to achieve a goal (such as SELECT statements in an SQL environment), rather should maintain the complete lifecycle of the mission itself, and this includes:

1. The plans and their evolution from the start of the mission to the final stage (mission completion).
2. Data and states of the mission are preserved.
3. Information about which agents are working in the mission at various states is preserved.

We believe this is important because:

(a) If the mission failed in its first run then it can be re-executed from the point just before the failure occurred. Normally, when a failure has occurred, the planner modifies (that is not re-plan) the current plan and gets it re-executed. Since the new plan has some similarities with the previous one then by avoiding the re-execution of tasks that were successfully executed in the previous run, we can make the system to achieve the goal faster and hence economically (because we do not need to re-execute those completed task again).

(b) The history of the mission being executed (included its successes and failures) is too valuable to be disposed. This is because such information might contain some important knowledge/information that the planner can use in future planning so that it can generate a better plan.

Our mission object comprises three major components: Task Decomposition Diagram (TDG), Mission Data Space (MDS) and Mission Execution History (MEH). The TDG is the structural representation of a plan. It lays out all the tasks (and their inter-relationships) that need to be carried out for a mission. The MDS is the component that provides a place for the tasks to interchange their results during the execution. Finally, the MEH is the component that maintains the execution history and the mission's states and data of each run.

2.1 Task Decomposition Graph (TDG)

In order to respond to the user's request, a set of tasks needs to be performed. These tasks are normally inter-related, and typically, have to be executed in a certain sequence in order to obtain a valid result. Therefore, these tasks and their inter-relationships must be represented in a non-ambiguous manner for software agents to understand and perform the tasks as intended. This representation is called Task Decomposition Diagram (TDG). TDG is a Direct Acyclic Graph (DAG) where the nodes represent the tasks that need to be performed and the links represent the inter-relationships between tasks. Our formal full formal definition of the TDG is presented in [8], the following summary is presented to help the reader to understand the discussion on this paper:

Definition 1. A task *t* is a tuple of the form *(u, n, y, s, o)* where:

U is a set of unique IDs and $u \in U$,
N is a set of locations at which a task must be performed and $n \in N$,
Y is a set of task types, which is *{Primitive, Compound}* and $y \in Y$,
S is a set of task statuses, which is *{Completed, Pending, InProgress, Failed, Aborted, Assigned},*
O is a set of functions that a task will execute and $o \in O$

A task has a status *Completed* when it has been successfully executed, *Pending* when it has not been executed nor assigned to any agent, *InProgress* when it is being executed, *Failed* when it has failed being executed, *Aborted* when it is stop in the middle of execution and *Assigned* when it has been assigned to an agent but the agent has not started the execution.

A task will be typed as *Primitive* when it represents a basic action that an agent can perform directly, and will be typed as *Compound* when:

(i) It is a complex task that can be broken down into a number of smaller tasks which can be primitive tasks and/or compound tasks
(ii) It contains some complex operational constructs, such as merging or filtering intermediate results during the execution.

A link between two tasks represents the relationship that holds between those two tasks, and its definition is as follow:

Definition 2. A link l is a tuple of the form (t_i, t_j, q), where

Q=*{Includes, DependsOn}*, T is a set of tasks and $i \in \mathbb{Z}^+, j \in \mathbb{Z}^+, q \in Q, i \neq j, t_i \in T, t_j \in T$

A link attribute *DependsOn* represents the dependencies that exist between tasks. If task A depends on task B, then task B must be completed first before task A begins. In other words, this attribute set the priori between tasks. The priori between tasks exists because of several reasons such as time or limit constraints, data dependencies to name a few. The attribute *Includes* on the other hand, captures not only the priori relationship between tasks but also the inclusion. For instance, a task of "building a house" will include tasks of "set building foundation", "erect building frame", etc. and hence it is logical to perform the foundation laying and frame establishment first before the house can be build.

Definition 3. *A Task Decomposition Diagram (TDG) is a DAG (Direct Acyclic Graph) = (T, L)*, where *T* is a set of tasks, *L* is a set of links.

Definition 4. A primitive task cannot depend on any other task(s), and this constraint is specified as follow:

Given $f_y : T \to Y$,

$$\forall (t_i, t_j, q) \in L, f_y(t_i) \neq primitive, i \in \mathbb{Z}^+, j \in \mathbb{Z}^+$$
$$and\ q \in Q$$
$$\forall (t_i, t_j, q) \in L, q = Includes, f_y(t_i) = Compound$$
$$\wedge f_y(t_j) = Primitive, i \in \mathbb{Z}^+, j \in \mathbb{Z}^+\ and\ q \in Q$$

Figure 1a shows the visual representation of the TDG. The circle nodes represent compound tasks while square nodes represent primitive tasks. *DependsOn* type is illustrated by using dashed-arrow-line, while a solid-arrow-line is used to illustrate the *Includes* type. The arrow dictates who depends on who. An arrow that is drawn from A to B means that A depends on B.

When a mission gets executed, the system actually executes the TDG. Executing all the available primitive tasks carries out the TDG execution. Available in this context means that those "pending" tasks. This can be regarded as bottom-up operation as well since primitive tasks are terminal nodes in the TDG. Primitive tasks cannot depend on any other task in our mode (refer to Definition 4) and hence they are executed in concurrent fashion.

A compound task is turned into a primitive task if and only if all its direct sub-tasks have been completely and successfully executed. Once the compound task has turned into a primitive type, it becomes a candidate for an execution in the next cycle of execution.

Figure 1 shows how a TDG is executed. First, tasks X, Y, T, U, V, W and Z are executed. On completion of the execution of task X, Y and W, task B's type will be changed to primitive. Similar to task B, task E becomes a primitive task because all of its subtasks are completed. Task D however, remains as a compound task because task E has not been completed yet. This situation is illustrated in Figure 1b. In Figure 1c, because tasks B and E have been completed, task D now becomes a primitive task and is ready to be carried out. Figure 1d shows the result after task D has completed. Finally, in Figure 1e, only a single task, i.e. task A, needs to be carried out. Upon completing this task, it is said that the mission has completed.

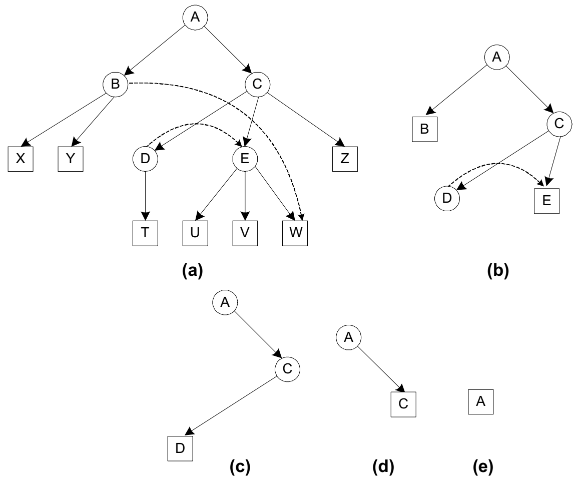


Figure 1: The Sequence of The TDG Execution

It should be noted that our TDG formal description above specifies that dependency links can only exist between compound tasks and between a compound task and a primitive task. When there is a logical dependency between primitive tasks or between a primitive task and a compound task, then the task on which the other depends must be converted into a compound task. By doing this, our model is guaranteed not to cause deadlock or live lock situations during execution

Figure 2 shows the conditions where we have to change the type of some tasks from the primitive into the compound type in order to avoid the deadlock and live-lock situations. In Figure 2a, task G depends on task B, and since both of them are primitives then task G has to be converted into a compound task as depicted in Figure 2b. In Figure 2c, a primitive task G depends on a compound task C, and hence it has to be converted into a compound task as illustrated in Figure 2d. In conclusion, we define that *a primitive task cannot depend on any other task(s)*.

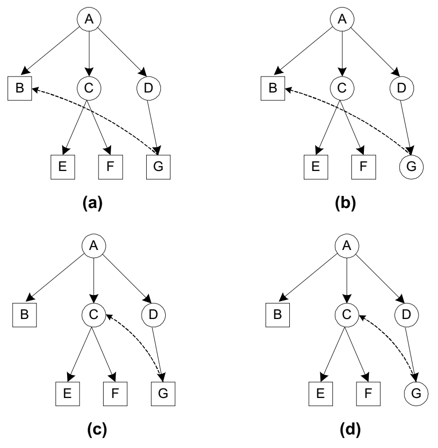


Figure 2: The TDG Task Type Conversion

A TDG is dynamic, that is, its structure can change at run-time. At the initial stage, a TDG represents the best possible plan a mission-planner can produce. It must be noted however, the planner can only predict what can happen without knowing if it will happen, i.e. it would be generally very difficult for a planner to produce an initial plan that is guaranteed to succeed. Depending on the dynamicity of the environment in which the plan is carried out, failures may occur. Hence, a mission must be dynamic for it to have a greater chance of succeeding. By that we mean that the system must be able to modify the plan at run-time. Modifying a plan means changing the TDG structure. Since the TDG is a DAG, then changing its structure will simply be adding and/or removing nodes and links. The discussion about how to change the TDG at run-time can be found in [6].

2.2 Mission Data Space (MDS)

Software agents execute tasks that are specified in the mission. In other words, every agent that is assembled at run-time is given one or more tasks to execute. When the tasks' locations are the same then the system will assign them to a single agent. When tasks in a mission are executed, they may or may not produce results. In turn, these results may or may not be needed by other agents who may either is currently working on other tasks, or simply is waiting for the results because those results are needed before the agents can continue working. The Mission Data Space (MDS) was created to cater such requirement. MDS is local to a mission, hence only agents that are working to a mission can read and write data to it. Other agents, although they are in the same community are not allowed to access the MDS.

We model the MDS to be local to a mission for the following reasons:

1. *Privacy*. In MAS shell, many missions can be run concurrently. Different users can executed the same missions (such as finding the best investment portfolio) at the same time. For this reasons we do not wish other agents than ours looking at our data.
2. *Security*. Most importantly, we cannot allow unfamiliar agents lurking around in our space trying to steal some piece of information

Definition 5. *Mission Data Space (MDS)* is defined a set of tuple of the form (u, r, t_S) where:

$u \in U$ and U is a set of task unique IDs (see Definition 1),
$r \in R$ and R is a set of output/result
t_S is a timestamp.

Based on the above discussion, it can be seen that MDS is the component that maintain the data of the mission at any time. MDS is not persistent by its own, in fact, its persistency is managed by the next module we are going to discuss, the Mission Execution History (MEH)

2.3 Mission Execution History (MEH)

The final component of the mission object is the Mission Execution History (MEH). The plan generated by the planner is the best possible plan that it can generate at that time. However, there is nothing preventing the failure of it at all. Our mission object, through its MEH component provides a mechanism for the system to suspend the mission, get the modified plan and re-execute the plan (recall, modifying the plan means modifying the TDG structure as mention in Section 2.1). MEH remembers the state and data at the time the mission gets suspended and re-execution is carried out from that point. Tasks that have been completed prior to the mission suspension will not be executed again unless the result is not valid anymore. For instance, if the current plan involves in some currency conversion then reusing a result from yesterday's execution may not be adequate. In this case the task must be re-executed.

Definition 6. *Mission Execution History (MEH)* is a set of tuple of the form $(r, t_S, d_{t_S}, p_{t_S})$ where:

$r \in \mathbb{Z}^+$ is the run number,
t_S is the timestamp at which the snapshot is taken
$d_{t_S} \in D$ and D is the MDS defined in Definition 5, is the snapshot of MDS at time t_S
$p_{t_S} \in P$ and P is the TDG defined in Definition 3, is the snapshot of the current plan/TDG at time t_S.

The run number is a positive-sequential number that represent how many times the mission had been run. From Definition 6 above, it can be seen how the state and data of the mission is maintained. The state of the intermediate data is maintained by capturing the content of the MDS while the state of the plan is maintained by capturing the current TDG. It should be noted that, since the TDG maintained the status of the tasks then the state of the tasks are also maintained.

When a mission gets suspended or stopped, an instance of MEH is created and stored within the mission object. The mission object can then be serialized to make it persistent. When the missions is resumed or re-run, then the last instance of MEH is loaded into the system and the mission execution is continued from the point where it suspended or stopped. We believe that such a notion is important. For instance, if the condition of the environment is impossible for the mission to be continued then it is a good idea to suspend the mission until the environment permits. However even, suspending the mission should not mean that when it is resumed, the system must start from the beginning again. These notions of mission suspension and on-demand agent assembling (which will be discussed in the next section) are some of the important features of our research project.

We acknowledge that maintaining the complete structure of TDG in each snapshot can make the mission object consumes unnecessary space. However, our main research focus at this stage is not about making the mission object as compact as possible. We also noted that the technique described in [5] can be used to minimize the size of the mission object.

3 eHermes: The Multiagent System Shell

The notion of assembling agents on-demand was first introduced in [6]. The main idea is to create agents only when they are needed. This is in contrast to systems that have a set of agents with predefined functionalities running and waiting for some tasks to be assigned to them. We regard such systems as inefficient systems especially because they might require substantial system resources. In our vision software agents should not exist when they are not needed and only get created when they are needed. We refer this notion to as *on-demand agent assembling*. Moreover, such creation and deletion of agents and the coordination of the agents (including newly created ones) should be automated. In our shell, agents are dynamically created based on the mission in-hand. A mission can be complex or simple, and naturally, a complex mission needs more computations, and hence, more agents might be needed to complete the request. For instance, a request to find the best investment portfolio for an individual is more complex than to find the best price for a home theatre system.

eHermes is the prototype system being developed as the proof of concept for vision described earlier. The conceptual view of eHermes as an agent-shell system is illustrated in Figure 3. eHermes has three major components, they are: *Mission Generator*, *Mission Planner* and *Agent Assembler*. Apart from these components, eHermes is equipped with several important repositories such as Ontology, Mission Repository and Agent Component Library (ACL).

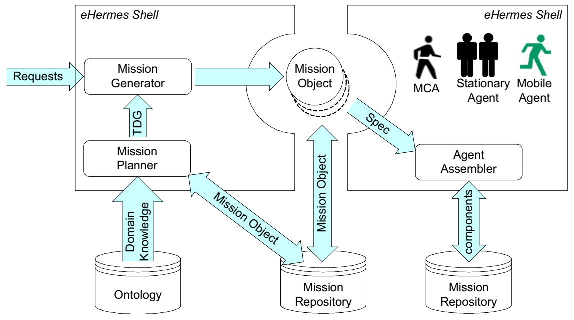


Figure 3: eHermes Conceptual Diagram

Mission object can be generated by the Mission Generator for a given request, or supplied from an external source, such as, human user or other system. A request comes from an actor, which again, can be a human user or other agent systems. An external source, which has a mission object already, can just inject this object into the system and hence removing the need regenerating it again. Such a notion is important for our approach because in our view, the mission must be suspend-able or stoppable at any time and can be resumed later. In other words, the mission is a pluggable object to the eHermes shell system.

Recall from the previous discussion that a mission object contains a plan/TDG. However, the Mission Generator is not equipped with the planning ability and hence must rely of the Mission Planner to supply it with a plan, which the system must execute to fulfill the request. The plan drawn by the planner is the best plan the planner could possibly produced at the time. Finding the best planning algorithm is beyond the scope of our project, however, any HTN-based planner [11] can be used.

When the planner makes a plan, it always check the mission repository in order to re-use the existing plan if there is any. If none is readily available, it will then decompose the request until the best final plan has been achieved. The final plan has the TDG-like structure. The Mission Planner will use appropriate domain knowledge in order to interpret the request correctly as some words have different meanings depending on the context in which those words were used. For instance, the word "surfing" would have a different meaning when the context is about the Internet as opposed to sports.

A TDG is returned back to the Mission Generator once the planning is complete. This TDG is then packed together with the structures to support MDS and MEH to become a mission object. The mission object is fed into the Agent Assembler component. The Agent Assembler is the component that is in charge for creating agents on the fly and at run-time. Agents created with this module can be a mobile or stationary agent. During the agent assembly, the Agent Assembler will use any reusable agent components, which is stored in the Agent Component Library (ACL).

When a mission object is pass on to the agent generator, a special agent called *Mission Control Agent (MCA)* is created. The MCA is a special agent because its main tasks are to:

(a) *Monitor and manage the execution of the given mission.* This involves in suspending and resuming the execution if requested or if the mission has failed.
(b) *Control the creation of worker agents.* This involves in specifying what needs to be created and when. Since it monitors the progress of the execution, it knows when and what type of agent is needed. Part of the specification it produces for the agent generator is the type of agent (mobile or stationary) and the task(s) this agent has to perform.
(c) *Coordinate the worker agents*. Agents that are working in the mission must be coordinated so that there no overlapping jobs, no race condition between agents and data from an agent is available when the other is needed.
(d) *Mediate between the actor and the system.* Inform the requestor about the status of mission execution at anytime.

The MCA does not execute the mission itself, rather through the help from a number of agents that gets created at run-time. These agents can be either mobile and/or stationary agents depending on whether the tasks need to be performed locally or remotely. Tasks are executed concurrently in a bottom-up fashion as presented in Section 2.1. By executing the tasks in this fashion, as opposed the one presented in [7], we are able to achieve a faster result, yet still simple to manage.

When the tasks are available to be executed (that is those with attributes are `primitive` and `Pending`), they will be inspected by the MCA. MCA will request the Agent Assembler to create agent(s) one for each task or a group of them. MCA uses the following strategies:

```
if At-Minimal-Cost is true then
{
  for all the tasks that are ready to be executed
  {
    if the task's location is local then  put into local-pool
      else   put into remote-pool.
    requests for creation of an agent for local-pool
    requests for creation of an agent for remote-pool
    instructs the agents to start executing the tasks
}
else
{
  for all the tasks that are ready to be executed
  {
    requests for creation of an agent for current task
    instructs the agent to carry out the task.
  }
}
```

As shown in pseudo code above, if the mission has to be run at the minimal cost then the system will group tasks together based on their locality. This group of tasks is then assigned to a single agent to be performed. If there is no cost restriction for the mission, the system generates agents for every single task in the mission and executes them concurrently.

Each of the worker agents that are created by the Agent Assembler has a Global Unique ID (GUID), which is returned to and maintained by MCA. MCA monitors the status and whereabouts of all the agents it requested.

The Collaboration diagram shown on Figure 4 illustrates the behaviour of eHermes. The diagram is simplified by removing some unimportant details yet it shows the behaviour of the system starting from accepting a request from an actor.

(1) An actor asks the system to solve a request. This request is sent to the Mission Generator.
(2) The Mission Generator calls one of the Mission Planner's services, called `plan()`, to get the best plan for the request supplied by the actor..
(3) Once the Mission Planner finished the planning, it returns a TDG to the Mission Generator
(4) Once the TDG is obtained, the Mission Generator creates a mission object.
(5) The Mission Generator uses the Agent Assembler's service called `createMCA()` to create the MCA. The mission object created in (4) is passed on to the Agent Assembler.
(6) Once the MCA is ready, it gets the TDG from the mission object.
(7) TDG execution starts now;
    (7.1) The MCA iterates the TDG and picks all those tasks that can be executed, i.e. those tasks that are pending and has not been assigned. Specifications for these agents are built at the same time.
    (7.2) For each of the specifications produced, an agent, which can be either mobile or stationary (depending on the specification) is built.
    (7.3) Once the agents are made, they are assigned with one or many tasks. The MCA sends a command "go" to instruct the agents to start executing the task(s). If the task(s) need to be performed remotely

then the agents travel to the target destination automatically before initiating the execution.

(7.4) When the result of previously executed task is needed, the agents send a request to MCA for the result.

(7.5) The MCA gets the result requested from the MDS in the mission object and returns it to the requestors.

The sequence of the executions presented above is repeated until all the tasks are completed. It should be noted that Figure 4 does not illustrate the interaction behaviour between the MCA and the actor, nor the behaviour of the system when there are one or more tasks have failed or have been aborted by the user. Such details are left out from the diagram in order to reduce the complexity of the diagram and hence ease the discussion.

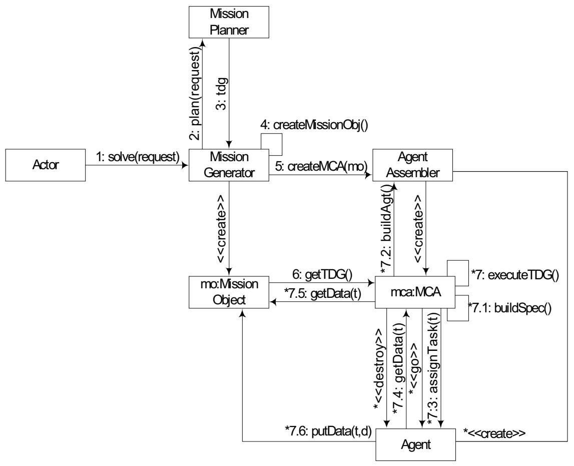
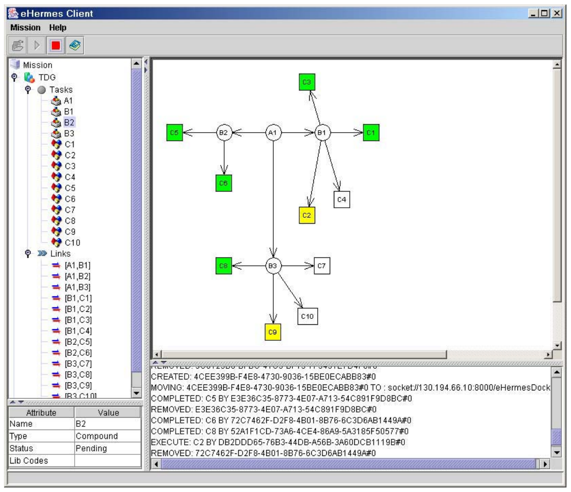


Figure 4: eHermes Behaviour Diagram

When a task has failed during the execution, the agent informs the MCA about the situation. The MCA then informs the planner and the actor about this situation. The actor may choose either to get the planner re-plan or abort the whole mission. If re-plan is chosen, then the planner will re-plan and send the modified part (not the whole plan) of the plan to MCA. This modified part is the delta (or the difference) between the old and the new plans. Once received, the MCA will change the internal TDG structure of the mission object as presented in [6]. Furthermore, the MCA will send various commands to the working agents either to ignore some previously assigned tasks (since now they are not par of the new plan) or abort some tasks.

eHermes is being implemented on top of the mobile agent toolkit called Grasshopper [9]. eHermes consists of two parts, they are the client and server sides. The client sides is a lightweight Java program which purpose is to give the user a visual interface to interact with the main engine. Java language was chosen because we would like out client side to be as portable as possible, so that it can be installed on small PDAs. Figure 5 shows the screen snapshot of eHermes client side.



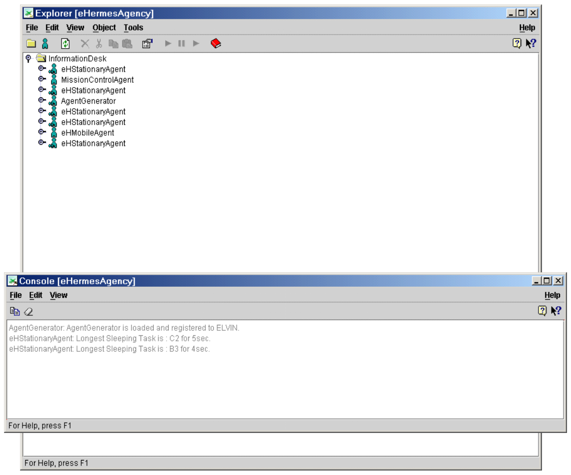
Attribute	Value
Name	B2
Type	Compound
Status	Pending
Lib Codes	

```
CREATED: 4CEE399B-F4E8-4730-9036-15BE0ECABB83#0
MOVING: 4CEE399B-F4E8-4730-9036-15BE0ECABB83#0 TO : socket://130.194.66.10:8000/eHermesDock
COMPLETED: C5 BY E3E36C35-8773-4E07-A713-54C891F9D8BC#0
REMOVED: E3E36C35-8773-4E07-A713-54C891F9D8BC#0
COMPLETED: C6 BY 72C7462F-D2F8-4B01-8B76-6C3D6AB1449A#0
COMPLETED: C8 BY 52A1F1CD-73A6-4CE4-86A9-5A3185F50577#0
EXECUTE: C2 BY DB2DDD65-76B3-44DB-A56B-3A60DCB1119B#0
REMOVED: 72C7462F-D2F8-4B01-8B76-6C3D6AB1449A#0
```

Figure 5: eHermes Client Screen Snapshot

The main window of the eHermes clients consists of 4 parts. The top left-hand side window shows the internal content of a mission, which includes the structure of the TDG, MDS and MEH. When the mouse is clicked on any of the item on the top left-hand side window, the bottom left-hand side window shows the details of the item of interest. For instance, if the item is a task, then the task's name, type, and status are shown. The top right-hand side window shows the visual structure of the TDG. The rectangle items represent the primitive tasks while the circles represent the compound tasks. Throughout the execution of the mission, these items are animated to indicate their status, such as executing, failed, aborted, pending or assigned. Finally, the bottom left-hand side window shows the log of the mission execution. It shows, which agents are created, destroy, moving, and executing.

The server side of the system contains a much simpler user interface, it lists the running agents which are dynamically created and destroyed at run-time. The Grasshopper agent toolkit was chosen because it is one of the few toolkits that supports mobile agents. Furthermore, since it has a Unix/Linux version we can perform testing on a number of hosts of different operating systems.



```
AgentGenerator: AgentGenerator is loaded and registered to ELVIN.
eHStationaryAgent: Longest Sleeping Task is : C2 for 5sec.
eHStationaryAgent: Longest Sleeping Task is : B3 for 4sec.
```

Figure 6: eHermes Server Screen Snapshot

4 Related Work

Decomposing a complex task into a hierarchical structure is not new; approaches such as TÆMS and Chandrasekaran's Task Structure [4] have been proven to be useful. With a similar view, the concept of agent generation has been approached before such as in DESIRE [3] and Zeus [10] to name a few.

Originally TÆMS was developed as a framework to analyse and design agent coordination mechanisms from the tasks, their interrelationships and environment point of view. TÆMS models the structure and interrelationship between tasks in a structure named Task Structure. Task Structure is similar to a plan. Task Structures can be super imposed one on another creating a family of plans. TÆMS is enriched by a set of predefined relationships and functions. TÆMS' Task Structure contains too many features that are not desirable in our project, such as multiple plans representation, and some predefined relationships to name a few. Because the mobility nature of our agents, carrying multiple plans is not desirable since it makes the size of the agent large and potentially has a monetary effect in a wireless.

In 1983, Chandrasekaran introduced a model he named Task Structure, which he used to model a knowledge based system from the point of view of tasks. Chandrasekaran's Task Structure is regarded is a simple task structure, which in fact is too simple for us to adopt. For instance, the Task Structure does not capture inter-relationships between tasks other than the "composed-of" relationship. Hence, there is no notion of priority (such as `A` must be performed before `B`) in the Task Structure. Furthermore, the Task Structure does not model any task attributes at all, which is regarded as one of the important keys for agent coordination in our model.

Brazier introduced a new notion in task decomposing by retaining other information such as control structure, knowledge structure and role delegation within the model itself [1]. This entirely can be done formally within a framework called DESIRE. Like other task decomposition techniques, DESIRE has a notion for compound and simple tasks, called *complex* and *primitive* tasks. However, not like others DESIRE has a capability of specifying the behaviour of complex tasks and data flows between those tasks. Like TÆMS, DESIRE is overly complex for the purposes of this project. However, DESIRE is able to specify the task behaviour and data flow. TDG allows the data flow based on Java's Java Space™ mechanism.

Zeus is a rapid agent development toolkit for multi-agent systems by providing the agent developers with a library of software components to use, such as for communication (KQML or ACL), coordination protocols, etc. While Zeus offers agent generation as well, it is different from eHermes in that Zeus' target users are developers while eHermes' are end users or even other agents. eHermes does not require its user to have any knowledge about how to build an agent system or even have some familiarities with the notion of agent. The user only needs to be able to supply the request in the format that eHermes understands and eHermes will take care of the agents' creation, execution and coordination.

Brazier in [2] promotes a notion called the *agent factory*. Brazier's main argument is that for a mobile agent to be effective, it must be able to move from one target to another in a heterogeneous environment, not just in a homogeneous environment. He points out that one possibility of achieving this is by moving the specification of that agent (and its data/state) as opposed to its actual code. The agent gets generated at the target host, supplied with the previous data/state and continues the execution at the host target. Similar to Zeus, agent factory requires the user to be agent-aware and have deep programming background.

5 Conclusions and Future Work

In this paper, we have presented the notion (and an implementation as a shell) of assembling agents (stationary or mobile) on-demand and at run-time, and then managing the execution of these agents, based on a mission generated by a Mission Planner. We note, however, that at this stage our approach relies on the ability of the Mission Planner to generate the best plan not only for the initial stage but also at later stage (run-time) should the plan needs to be modified or adjusted. Furthermore, we assume at this stage that the planner will take a reasonable amount time to generate a plan. Our research focus is not about finding a technique to generate a plan but rather on how to execute a plan (as a mission) by using a set of dynamically assembled agents. We have presented our task decomposition model, which we called Task Decomposition Graph (TDG) and the strategy for its execution. Such a treatment of tasks also supports task and component reuse across different applications. Our mission object model has also been presented along with the inter-agent communication and history mechanisms. Finally, we presented our prototype system eHermes, which is still actively being developed, and at this point, has most of the core functionalities implemented.

Future work includes the following:

(a) *Coordination*. So far, our agent coordination mechanism is simple, that is based on the geographical information (location). Tasks are grouped together based on their execution location. These tasks are assigned to a single agent if the mission should be a budget conscious one, multiple agents otherwise. From the hostname or IP address information specified within the tasks, eHermes searches the geographical information using tools such as GEOBytes.[1] However, such strategy still needs to be enhanced by including a cost-benefit analysis so that eHermes can provide an optimum strategy.

(b) *Quality of Results Over Time (QROT)* is defined as the degree of quality of a piece of information (result) over time. We observed that the results of a task execution might have quality that decays over time. A quality value of zero means that the result is no longer usable and hence the task needs to be re-executed. Simple examples for this notion are currency exchange or boiling water. If the task involves converting one currency to another then suspending a mission for a day is not desirable because by then the value of the conversion might not be so accurate or even not usable.

Acknowledgements.

Support for this project from the Australian Research Council (ARC) Linkage Project scheme and Microsoft Research Asia Grant LP0211384 is thankfully acknowledged.

References

[1] F. M. T. Brazier, P. H. G. v. Langen, J. Treur, N. J. E. Wijngaards and M. Willems, *Modelling A Design Task in DESIRE: The VT Example*, Faculty of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1995.

[2] F. M. T. Brazier, B. J. Overeinder, M. Van Steen and N. J. E. Wijngaards, *Agent Factory: Generative Migration of Mobile AGents in Heterogeneous Environments*, *Proceedings of The 2002 ACM Symposium on Applied Computing (SAC 2002)*, ACM Publisher, Madrid, Spain, 2002, pp. 101-106.

[3] F. M. T. Brazier, J. Treur, N. J. E. Wijngaards and M. Willems, *Formal Specification of Hierarchically (De)Composed Tasks*, in M. Musen, ed., *The 9th Knowledge Acquisition for Knowledge Based Systems Workshop*, University of Calgary, 1995.

[4] B. Chandrasekaran and T. R. Johnson, eds., *Generic Task and Task Structures: History, Critique and New Directions*, Springer -Verlag, Berlin, 1993.

[5] G. T. Jayaputera and K. E. Cheng, *SoftEAM: A Design History and Justification Maintenance Tool*, Australian Computer Journal, 26 (1994), pp. 124-133.

[6] G. T. Jayaputera, S. W. Loke and A. Zaslavsky, *Mission Impossible? Automatically Assembling Agents from High-Level Task Descriptions*, in J. Liu, B. Faltings, N. Zhong, R. Lu and T. Nishida, eds., *The 2003 IEEE/WIC International Conference on Intelligent Agent Technology (IAT 2003)*, IEEE Computer Society, Halifax, Canada, 2003, pp. 161-167.

[7] G. T. Jayaputera, A. Zaslavsky and S. W. Loke, *Mission-Based Multiagent System for Internet Applications*, in L. Camp, J. Filipe, S. Hammoudi and M. Piattini, eds., *Proceedings of the 5th International Conference on Enterprise Information Systems*, Escola Superior de Tecnologia do Instituto Politecnico de Setubal, Angers-France, 2003, pp. 232-237.

[8] G. T. Jayaputera, A. Zaslavsky and S. W. Loke, *Run-Time Mission Evolution in Mobile Multiagent Systems*, *The 2004 IEEE/WIC/ACM International Conference on Itelligent Agent Technology (IAT2004)*, IEEE Computer Society, Beijing, China, 2004, pp. To Appear.

[9] T. Magedanz and C. Bäumer, *Grasshopper - A Universal Agent Platform Based on OMG MASIF and FIPA Standards*, http://www.cordis.lu/infowin/acts/analysys/products/thematic/agents/ch4/ch4.htm, 1999.

[10] H. S. Nwana, D. T. Ndumu, L. C. Lee and J. C. Collis, *ZEUS: A Toolkit and Approach for Building Distributed Multi-Agent Systems*, in J. M. Bradshaw, ed., *Proceedings of the Third International Conference on Autonomous Agents (Agents '99)*, ACM Press, Seattle, USA, 1999, pp. 360-361.

[11] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.

[12] K. Sycara, M. Paolucci, M. van Velsen and J. Giampapa, *The RETSINA MAS Infrastructure*, Autonomous Agents and Multi-Agent Systems, 7 (2003), pp. 28-49.

[1] http://www.geobytes.com/IpLocator.htm?GetLocation