# Mission Impossible? Automatically Assembling Agents from High-Level Task Descriptions

Glenn Jayaputera, Seng Loke and Arkady Zaslavsky
School of Computer Science and Software Engineering, Monash University, Melbourne, Australia
*gtj@csse.monash.edu.au, Seng.Loke@csse.monash.edu.au and Arkady.Zaslavsky@csse.monash.edu.au*

## Abstract

*In this paper, we present our notion of automatically assembling agents on-demand given high-level task descriptions. Our approach is based on the concept of the mission, which is represented by a hierarchical structure called Task Decomposition Diagram (TDD). Tasks in TDD are specified using a task specification language, which defines the functionality and possible implementation of those tasks. We discuss several strategies for executing the TDD, for assembling agents based on the TDD, and present the architecture of our prototype system called eHermes.*

## 1. Introduction

A software agent is a system capable of autonomous actions in a dynamic and unpredictable environment. A software agent is typically associated with a mission that it has to carry out autonomously, perhaps in coordination with other agents in a multi-agent system.

There are many agent systems being developed so far, such as MAgNET [1], InfoSleuth [2], Chayani [3], BIG [4] and IntelliShopper [5], to name a few. However, these software agent systems are built for specific purposes and do not allow much flexibility at run-time. For instance, Morpheus [6] and IntelliShopper [5] were developed for on-line shopping while systems like InfoSleuth and BIG were developed to perform information extraction from heterogeneous on-line information sources. Because of the specialized nature of these systems, they were built with a fixed number of agents with fixed functionalities. With our approach, not only is an initial set of agents automatically assembled for an application based on a high-level task description, but also new agents can be dynamically created at run-time if needed.

Constructing multiagent systems (including the individual agents and the coordination mechanisms) is generally an arduous task and requires sophisticated software engineering methodologies and tools. However, 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 assemble a multiagent system for the application from a high-level description of the mission.

Based on this hypothesis, we explore in this paper, a novel notion, which we believe, will extend current software agent technology to allow wider applicability of software agents. Our proposal is to create agents on-demand and at run-time. The number of agents and their functionality will depend on the complexity and nature of the user's initial request itself. 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 [7] in that eHermes goes further in what it attempts to automate, in bridging the gap between a user's request and a system that fulfills that request.

The rest of this paper is organized as follows. Section 2 presents our on-demand agent-assembly notion. Section 3 discusses the definition of the mission. Section 4 presents the technical discussion about our task decomposition diagram and the task specification language we use to specify the tasks in the diagram. Section 5 presents the architecture of our prototype system called eHermes, and finally, Section 6 concludes this paper with a brief discussion about future work for this project.

## 2. Assembling Agents On-Demand

The notion of assembling agents on-demand was first introduced in [8] and [9]. 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.

Such systems are not efficient especially because they might require substantial system resources. The 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 notion, agents are dynamically created based on the request sent to the system. A request can be complex or simple, and naturally, a complex request needs more computations, and hence, more agents might be needed to complete the request. For instance, a request to find the best life insurance policy available might be more complex than to find the best price for a pair of shoes.

Furthermore, from the software engineering point of view, the ability to assemble agents on demand automatically is much simpler than low-level programming of agents using C or the Java programming language, even if some agent libraries are employed.

## 3. Mission

Requests from the users are informal and ambiguous, and hence must be represented in a way that a system can understand and work with. We name this representation a *Mission*. Mission is the main concept behind our vision and approach. A Mission is defined as *a purpose or goal an agent has to accomplish to satisfy its owner's request*.

A request that is supplied by a user is converted into a mission so that the agent can understand. In carrying out the mission, the agent performs a number of tasks. Hence, a mission can be elaborated into *a set of tasks that the agent must perform* [8]. If all the tasks specified within that mission have been completed, it is then said that the mission has been completed.

A mission is generated by a planning system. Planning systems such as GPGP [10] can be used. However, since planning is not this paper's focus, they are not described in detail.

A mission is a *living object*, that is, it is not just a static representation of tasks to perform but is changing dynamically during run-time, of course with the change in a constrained way that we do maintain the initial intention of the user's request. Tasks are added or deleted and results are collected, combined and deduced at run-time.

## 4. Task Decomposition Diagram (TDD)

In eHermes' mission, there are two types of tasks, namely: *compound* and *primitive* tasks. A primitive task is defined as a basic action that an agent can perform directly. A compound task, on the other hand, is regarded as a task that is composed of primitive tasks and/or other compound tasks. In order to understand this task-type definition, consider the following scenario. Let us assume that our agent is a robot that can move from one location to another. In this scenario, the ability to move the robot's right and left foot forward is considered to be the basic

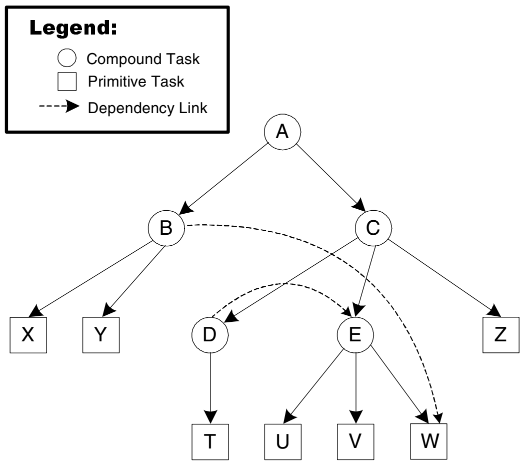

**Figure 1: Task Decomposition Diagram**

action it can perform. In a mobile agent environment, a primitive action might be its ability to "move" from one host to another.

Tasks in a mission are inter-related to each other by dependencies and external constraints such as time and resources. A task might not be able to be performed until some other tasks are completed, that is, that task depends on the completion of other tasks. On some other occasions, a task might not be able to be completed because "it is not the right time yet" (time constraint). For instance, for an auction-bidding agent to successfully bid for as minimal price as possible, it should wait until it is near the closing time. If the agent bids too early then other agents or users might overbid our agent[1] resulting in the final price being too expensive.

Therefore, it is identified that in order to "instruct" an agent to do something, we must provide it with a mission and that mission must be in a form that the agent can understand and hence be able to execute.

A mission in eHermes is represented as an attributed Direct Acyclic Graph (DAG) and is called *Task Decomposition Diagram (TDD)*. Tasks are represented as nodes while *compose-of* and *depend-on* relationships between tasks are represented as links. To illustrate this notion, consider the diagram in Figure 1.

In Figure 1, circle nodes represent compound tasks while square nodes represent primitive tasks. A compound task is composed of one or more tasks and these tasks can be composed of one or more compound or primitive tasks. The root node; A is regarded as the mission itself. A is said to be achieved if and only if tasks B and C are completed. In turn, task C is completed if tasks D, E and Z are completed.

Dependencies are represented by dashed-arrows in a task decomposition diagram. Task T1 is said to be

[1] It is assumed in this context that automatic bidding is not allowed

dependent on task T2 if there is a dashed arrow drawn from T1 to T2. Taking this notion into account, we can say that in Figure 1, task B depends on W and D depends on E. This means that task B cannot be executed until W has. Subsequently, task D cannot be carried out until E has completed.

The Task Decomposition Diagram (TDD) is similar to the TÆMS' Task Structure [11] in that they represent a hierarchical abstraction of tasks to perform and the inter-relationships between those tasks. However, TDD is simpler compared to TÆMS. For instance, TDD does not contain a triggering mechanism such as the "Enable" link in TÆMS. Furthermore, TDD does not contain quality attribute and hence, functions associated with this attribute.

Unlike TÆMS, TDD does not represent a family of task structures but only a single task structure. This is because the task structure in TDD is the actual plan that is going to be used at run-time, as opposed to being used at the elaboration or testing-out phase.

Furthermore, unlike TÆMS, TDD is dynamic. A TDD can grow and shrink during run-time. As the mission execution progresses, some tasks are completed and those completed tasks are removed from the TDD. The task structure that the TDD represents is the best possible structure that the planner can generate at the time. This however, may or may not be feasible at run-time. Tasks that look to be feasible and achievable at design/planning time might not be so at run-time. Therefore, new tasks must be added to the TDD. This mechanism allows the TDD to cater for "alternative-ways", but changes are performed at run-time and hence dynamic.

Furthermore, we argue that although to have a family of task structures is useful, since only a single task structure is going to be used at run time then it is not practical to represent them all in the TDD. We believe that the planner must choose the best one for the mission.

The planner can only predict what can happen at run-time without knowing if it will happen. Hence, the ability of the TDD to represent a number of task structures will not increase the success rate of a mission. This is because there is a possibility that all of them cannot be used at run-time. Therefore, instead of having `n` number of task structures in the TDD, we allow the TDD to change at run-time. By allowing the TDD to change at run-time the success rate of the mission is increased. As soon as the current TDD does not fit, a message will be sent to the planner along with all the run-time conditions and information. The planner then will re-plan the mission and send the delta between the new and old TDDs to the agent.

Since the TDD is represented with nodes and arcs, then changing the TDD will simply become adding and/or removing nodes/arcs. For instance, assume that the planner re-design the TDD presented in Figure 1 to become the one shown in Figure 2.

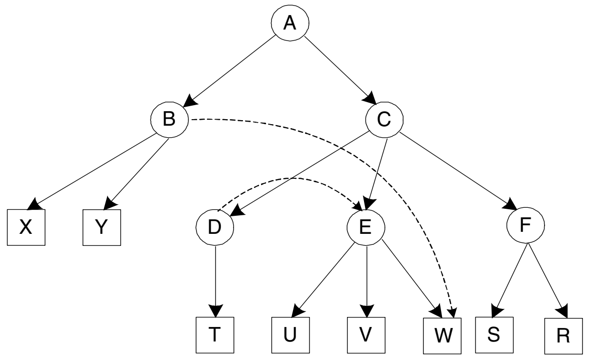

**Figure 2: A New Mission Representation**

Recall that TDD is defined as a graph; *G=(V,E)* where *V={v1, v2, v3,…}* is a set of vertices and E=*{(v1,v2), (v2,v3),…}* is a set of edges. The TDD shown in Figure 1 is then can be textually represented as below:

```
G1={A, B, C, D, E, T, U, V, W, X, Y, Z, (A,B), (A,C), (B,X), (B,Y), (B,W), (C,D), (C,E), (C,Z), (D,E), (D,T), (E,U), (E,V), (E,W)}
```

And the new mission shown in Figure 2 can also be represented in the similar manner as shown below:

```
G2={A, B, C, D, E, F, R, S, T, U, V, W, X, Y, (A,B), (A,C), (B,X), (B,Y), (B,W), (C,D), (C,E), (C,F), (D,E), (D,T), (E,U), (E,V), (E,W), (F,S), (F,R)}
```

Note that, for the simplicity of this discussion, we did not use different notation to differentiate the textual representation of compose-of and depend-on links. One can easily use square brackets for dependency links to differentiate them from compose-of links, which use round brackets.

From those simple text representations, a delta between the two TDD can be deduced and they are illustrated below:

```
-Z, -(C,Z), +F, +(C,F), +(F,S), +(F,R)
```

Where the minus sign denotes "remove" from the original and the plus sign denotes "add" to the original. Using this technique, we are able to modify the TDD at run-time and hence add dynamism to the mission. It should be noted that by performing such action we are practically performing the logical OR operation except in our case it is dynamically performed.

The specification of tasks in TDD is defined using a task description language called Durra [12]. Durra describes the functionality of tasks, input/output parameters the tasks take, and their interactions at the logical and abstract levels not at the programming level. This is suitable with TDD since TDD describes a mission at a high level of abstraction as well. The task specification is later converted into the actual

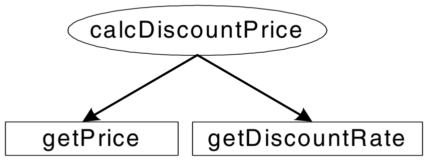

**Figure 3: Simple Scenario**

programming code by a module called *Agent Generator*, which will be presented in Section 5.

A task description in Durra is composed of four components, they are: interface, behavioral, attributes and structural information. The interface component specifies the data movement to and from the task, as well as the type of the data. The behavioral component specifies the functional and timing information of the task. The functional information consists of the pre and post-conditions of the data movement (that is incoming and outgoing data). The timing information specifies the timing expression of the task. The attribute component specifies miscellaneous properties of a task. This attributes is in the format of `AttributeName=Value` pair. The structural information provides the internal structure of the task. Readers are requested to refer to [12] for a more detailed discussion on Durra.

The following is a very simple scenario for calculating a discounted price of an item. The discount rate (in percentage) and the item's price are obtained from the user. It should be noted here that how the user entered these figures is not important (it can be done via a simple text-based or complex GUI-based program). The discounted-price is then simple calculation of `price-(price*discount)`. A TDD representation of this simple scenario is illustrated in Figure 3.

Figure 3 shows that the compound task `calcDiscountPrice` is decomposed into 2 primitive tasks, they are `getPrice` and `getDiscountRate`. Although structurally it is clear, however, the logical and behaviour description of those tasks are unkown. The task specification below remove those ambiguities.

```
task getPrice
  ports
    input:  String sLine;
    output: int nPrice;
  attributes
    library="/var/lib/eHermes/AGL/getprice";
end getPrice;

task getDiscountRate
  ports
    input:  String sLine;
    output: float fRate;
  attributes
  library="/var/lib/eHermes/AGL/getDiscountRate";
end getDiscountRate;

task calcDiscountPrice
  ports
    output: nDiscountedPrice int;
  components
    t1: task getPrice;
    t2: task getDiscountPrice;
  structure:
    return: t1 * (1-(t2/100));
end calcDiscountPrice;
```

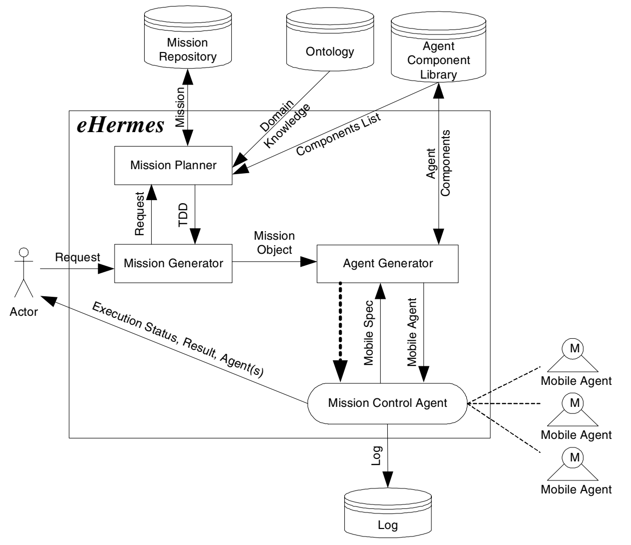

**Figure 4: eHermes Architecture**

The "library" attributes of the `getPrice` and `getDiscountRate` above specify the location of the actual implementation of these tasks. These implementations will be loaded and executed at run-time (that is when an agent perform those tasks). The "structure" component of the `calcDiscountPrice` task, specify the actual operation that needs to be performed by this task, which is `t1*(1-(t2/100))`.

## 5. eHermes: On-Demand Agent Assembler System

eHermes is the prototype system being developed as the proof of concept for the "on-demand agent assembling" vision that we propose. The architecture of eHermes is presented in Figure 4.

eHermes is composed of two main components, they are: *Mission Generator* and *Agent Generator*. Mission generator is the front-end module whose task is to generate a mission. Agent Generator is the module, which is in charge of generating agents during run-time.

A request from a user is sent to the mission generator, which will produce a mission for that request. Mission generator, however, does not have planning capabilities and hence rely on a module called *Mission Planner* for any planning issues. Finding the best planning algorithm is beyond the scope of eHermes project, and hence we use a planning concept called GPGP [10] for our exercise.

Once a request arrived at the mission generator, it forwards that request to the mission planner. Mission planner generates a plan, in the form of a TDD for this request. Plan or sub-plans may be freshly generated or reused from the one stored in the mission repository. 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.

Furthermore, the domain knowledge plays an important role during the planning process. The domain knowledge is used to retrieve a list of all available agent components that can be used for the context in interest. Each domain has a list of all available components that can be used. The component(s) that are used in a plan will be referred to via the "library" attribute part of the task specification described in the previous section. It should be noted however, that the implementation part of each agent component library is not used during the planning process but later during the execution of the tasks.

Once the planning is completed, a TDD is returned to the mission generator. This TDD contains the tasks description as well. The mission generator then constructs a mission object with the TDD embedded in it. This mission object is then forwarded to the agent generator.

Agent generator creates a special agent called *Mission Control Agent* (MCA) in order to manage and coordinate the execution of the mission. Once MCA is alive, it will execute the mission by executing the tasks specified in TDD. The MCA carefully examine the TDD structure and executes tasks without breaking any dependencies specified in TDD. Primitive tasks are executed first, either serially or in parallel whichever best for the mission.

MCA does not execute the whole mission by itself especially when the mission is complex. Instead, it requests other agents (mobile or stationary) to be created for it. Requests for agent creation are forwarded to Agent Generator, which will create agents for MCA. MCA forwards the specification for the agent it needs to the agent generator. This specification includes the itinerary the agent must go and visit (in case of mobile agent) as well as the task specification described in the previous section. The value of the library attribute described in the task specification is used during the assembling process by the agent generator. Agent Generator will locate the actual implementation code of this task pointed to by the library attribute, load the code and embed the code into the agent it is creating. This is analogues to giving a functionality to an agent which otherwise will be a dumb agent. For instance, the task `getPrice` specified in the previous section will dictate the agent about what to do once it is alive.

One way to execute the TDD is to sequentially execute the tasks without breaking the dependencies that exist between them. Since TDD is a DAG then this can be achieved by executing all the primitive-tasks that do not depend on others. To successfully do this, the TDD needs to be converted into a sequence of tasks yet still preserving the dependencies between them. Topological Ordering is used to achieve this. A topological order of a graph $G=(V,E)$ is a linear ordering of all its vertices such that if $G$ contains an edge $(u,v)$, then $u$ appears before $v$ in the ordering. The algorithm computes a topological order of a graph by maintaining a list of nodes with zero in-degree. After deleting these nodes from the graph, other nodes will have zero in-degree. The algorithm can be summarized as follow:

```
Topological-Ordering(G)
1   n ← 0
2   for each v in V
3     do {
4       toporder[n] ← v
5       remove v from V
6       n ← n+1
7     } iff indegree(v) = 0
8   return toporder
```

Using this algorithm, the TDD presented in Figure 1 can be sequenced to become:

`{A, B, X, Y, C, Z, D, T, E, U, V, W}`

By executing this series of tasks from right to left (that is from `W` to `A` in that list), it is guaranteed that the dependencies between those tasks hold.

Executing tasks sequentially is certainly not an optimal solution and hence the following two strategies are used to achieve better results. These two strategies are:

1. *Multi-itineraries and Tasks.* Assigning a task to an agent to be executed remotely requires the agent to move to the desired place before executing the task it is assigned. This will be an expensive exercise if there are a number of tasks that are be executed at the same location because the agent has to travel `n` times for `n` number of tasks, where $\exists$`n:1..`$\infty$. To avoid these repetitive and unnecessary movements, MCA selectively examines the tasks in TDD. Tasks that can be executed in one location are grouped together and assigned to a single agent. Hence there will be only one movement made by the agent. Once the agent arrived at the destination, it executes those tasks either sequentially or in parallel. The result of the executions are stored as a pair against the tasks themselves and brought

back to MCA once the executions are completed. The diagram in Figure 5 illustrates the concept.

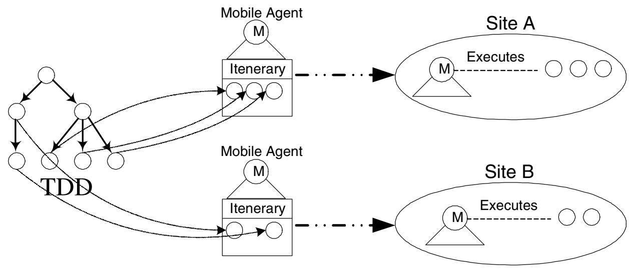

**Figure 5: Multiple Task Assignments**

Referring back to the simple example depicted in Figure 3, MCA could assign those three tasks to a single agent, which will execute them and bring back the discounted price to MCA. It should be noted however that there is no ordering between tasks `getPrice` and `getDiscountRate`, and hence should this become a necessity then a dependency link should be used between them.

2. *Remote Control Agent*. When the mission is complex, it necessary to create a miniature of MCA and give it a subset of TDD for it to execute and coordinate. This is notion behind Remote Control Agent (RCA). RCA is actually a remote MCA that is working on a remote site on behalf of the main MCA. Communication is hence only occurred between RCA and MCA, not between the agents created by RCA and MCA. In the environment where cost is based on the date being transferred, this reduced communication traffic can reduced the overall cost. RCA has all the attributes the MCA has, including the coordination strategies. This means that RCA is autonomous to MCA. RCA does not communicate with other MCA in the system, but only to the one that creates it. In fact from the user point of view, there is no agent called RCA but only MCA. To illustrate this point, consider the diagram presented in Figure 6. From that figure, it is shown that RCA creates agents locally, assign task(s) to them, launches those agents and coordinate as well as monitor the task executions. Once the submission has been completed, RCA will return to the site where MCA resides and gives the result back to MCA. MCA will combine the results from a number of RCAs (if necessary) before presenting the final one to the user.

eHermes is still being actively developed and implemented. The current implementation of eHermes is based on the Grasshopper [13] agent toolkit. At the moment, we have implemented some of the basic functionalities presented in this paper. Figure 7 is the screen snapshot of eHermes running on one of the nodes. Two MCAs are currently running. These two MCAs are serving requests from two different users.

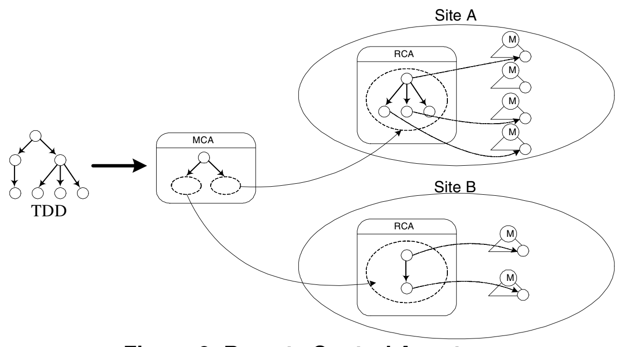

**Figure 6: Remote Control Agent**

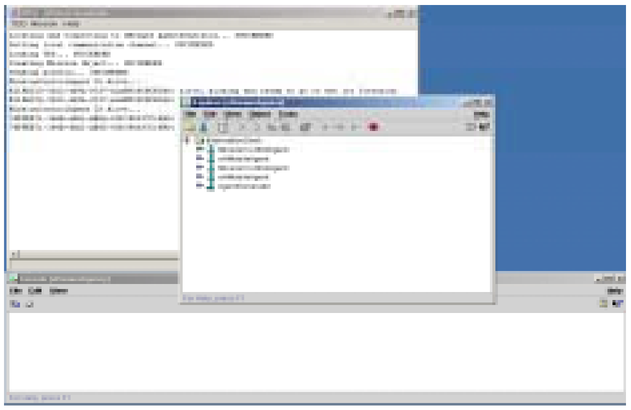

**Figure 7: eHermes Running on One of The Nodes**

## 6. Conclusion

We have presented in this paper, a notion and an approach to assembling agents on-demand, based on the mission generated by the mission planner. This is in contrast to Blackboard-based multiagent systems where a fix number of agents with fixed functionalities are running all the time waiting for a task(s) to be given to them. We argue that such a strategy is inefficient, especially when missions are changing all the time, depending on the request(s) given by the user(s) to the mission planner. For instance, keeping a bidding agent all the time is not efficient when the user only bids once or twice in a week.

We have presented our hierarchical task structure, called Task Decomposition Diagram (TDD), which is used to represent a mission. TDD is used to represent the inter-relationships between tasks within a mission so that a special control agent, called Mission Control Agent (MCA) can execute those tasks without breaking any dependencies between them. TDD is also used by MCA to coordinate the task execution. TDD is dynamic; it can shrink and grow at run time. We noted that such dynamism comes with some side effects, that is, it can slow down the mission execution. Furthermore, we assume that at this stage that the mission planner is a good planner in generating the initial TDD.

A language called Durra is used to describe the behaviour of a task. Our concept of loading an agent's functionality via a collection of components, called Agent Component Library (AGL) is also presented.

The architecture of our prototype system called eHermes, has also been presented and discussed in this paper. The discussion about how a TDD is executed and two strategies to distribute the MCA's load are presented.

Future work for this paper falls into several categories, they are:

1. Enhancements on the optimization logic and algorithms to allow sub-mission slicing for parallel execution.
2. Formalization of the TDD, including but not limited to the ability to express other constructs such as repetition (looping) or conditional branching.
3. Run-time mobile coordination based on the TDD attributes that get instantiated during run-time.
4. Possibility of replacing Durra with a light-weight task specification language. We found Durra too heavy-weight for our purpose because it was developed for specifying tasks for parallel computing.
5. Designing more TDDs for the information-related domain, involving information retrieval, information search, and data mining.

## Acknowledgements

Support for this project from the Australian Research Council (ARC) Linkage Grant LPO211384 is thankfully acknowledged

## References

[1] P. Dasgupta, N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "MAgNET: Mobile Agents for Networked Electronic Trading," *IEEE Transactions on Knowledge and Data Engineering, Special Issue on Web Applications*, 1999.

[2] R. J. Bayardo Jr, W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk, "InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic environments," in *Reading in Agents*, M. N. Huhns and M. P. Singh, Eds. San Francisco: Morgan Kaufmann Publishers, Inc., 1998, pp. 205-216.

[3] P. S. Dutta, S. Debnath, and S. Sen, "A Shopper's Assistant," presented at Proceedings of the Fifth International Conference on Autonomous Agents, Montreal, Quebec, Canada, 2001.

[4] V. Lesser, B. Horling, A. Raja, X. Zhang, and T. Wagner, "Resource-Bounded Searches in an Information Marketplace," *IEEE Internet Computing*, vol. 4, pp. 49-58, 2000.

[5] F. Menczer, W. Nick Street, N. Viswakarma, A. E. Monge, and M. Jakobsson, "IntelliShopper: a Proactive, Personal, Private Shopping Assistant," presented at Proceedings of the first international joint conference on Autonomous agents and multiagent systems, Bologna, Italy, 2002.

[6] J. Yang, H. Seo, and J. Choi, "MORPHEUS: A More Scalable Comparison-Shopping Agent," presented at Proceedings of the Fifth International Conference on Autonomous Agents, Montreal, Quebec, Canada, 2001.

[7] H. S. Nwana, D. T. Ndumu, L. C. Lee, and J. C. Collis, "ZEUS: A Toolkit and Approach for Building Distributed Multi-Agent Systems," presented at Proceedings of the Third International Conference on Autonomous Agents (Agents '99), Seattle, USA, 1999.

[8] G. Jayaputera, A. Zaslavsky, and S. W. Loke, "A mission-based Multiagent System for Internet Applications," presented at The Fifth International Conference on Enterprise Information Systems (ICEIS 2003), Angers-France, 2003.

[9] G. Jayaputera, O. Alahakoon, L. Cruz, S. W. Loke, and A. Zaslavsky, "Assembling Agents On-Demand for Pervasive Wireless Services," presented at The Second International Workshop on Wireless Information Services (WIS 2003), Angers-France, 2003.

[10] K. S. Decker and V. Lesser, "Designing a family of coordination algorithms," presented at The 1st International Conference on Multi-Agent Systems (ICMAS-95), Seattle, WA, 1995.

[11] K. S. Decker, "TÆMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms," in *Foundations of Distributed Artificial Intelligence*, G. O'Hare and N. R. Jennings, Eds.: John Wiley & Sons, Inc., 1996, pp. 429-447.

[12] M. R. Barbacci, D. L. Doubleday, M. J. Gardner, R. W. Lichota, and C. B. Weinstock, "Durra: A Task-Level Decription Language Reference Manual," Carnegie Mellon University, Pittsburgh, Pennsylvania CMU/SEI-91-TR-18, Dec 1991 1991.

[13] 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.