Mission-flow Constructor

A Workflow Management System Using Mobile Agents

A thesis
submitted to the Faculty
in partial fulfillment of the requirements for the
degree of

Master of Science
in
Computer Engineering
By
V. Shankaran Sundaram

Thayer School of Engineering
Dartmouth College
Hanover, New Hampshire
May 2000

Examining Committee:

Prof. George Cybenko: ______________________

Prof. Robert S Gray: ______________________

Prof. Clayton Okino: ______________________ ______________________

Dean of Graduate Studies

Thayer School of Engineering

Dartmouth College

Mission-flow Constructor

A Workflow Management System Using Mobile Agents

V. Shankaran Sundaram
Master of Science

Abstract

Developing code for the execution of a distributed, dynamic workflow requires significant effort and hence it becomes necessary to build tools that enable the creation and execution of such workflows. Compelling arguments have been made for the implementation of workflow management systems using mobile agents [CGN96, MLL97]. Mobile agents are autonomous pieces of code that can migrate under their own control from one machine to another within a heterogeneous network. Mission-flow Constructor (MfC) is a workflow management system built on the D'Agents mobile agent system [GCKR96]. Like its predecessor Mobile Agent Construction Environment (MACE) [Sha97], MfC uses the concept of visual languages and further abstracts the process of building a workflow. Agents generated by MfC are small and migrate only once. These agents hence make more optimal use of network resources than those generated by MACE. MfC generated agents also use improved communication means and incorporate some basic fault tolerance mechanisms. A set of primitive constructs that encapsulate commonly used topologies has been defined to make easier the process of workflow definition. A workflow specified using the GUI and associated annotation process is compiled to a set of D'Agents agents by making use of the visual depiction and the code fragments that define the individual modules. MfC then launches these agents to execute the various tasks associated with the workflow specified by the user.

Acknowledgements

First and foremost, I would like to thank my advisor, Prof. George Cybenko, for his infinite patience and understanding, his kindness and generosity, his contagious enthusiasm, and his gentle way of teaching me life's harder lessons. I would also like to thank Prof. Robert S. Gray for his invaluable help in teaching me about the D'Agents system and for spending so much his valuable time proofreading this thesis. I am forever indebted to my friends (and incidentally, my colleagues) in The Laboratory for Total Global Domination for whom my thanks cannot be said in any order but alphabetical. To Daniel Bilar, for his prodigious talent for coming up with quotable phrases, for his unique, and more often than not, hilarious perspectives, for teaching me the importance of being honest with myself, for the many, many times he stood outside Thayer with me so that we could take a break, and for always being a gentleman. To Brian Brewington, for being a prince among men and having a heart as big as a house, for his counsel and support in trying times, for his zany humor that could not be repressed, for being one of the reasons I have occasionally made a positive score in BZFlag, and for being the person I could always turn to for help. To Daniel Burroughs, for his gentle and unassuming nature, for always being able to make conversation, for his company over countless lunches, for his cheerful presence, for his innumerable witticisms, for being my partner in crime each time the word "battle" was heard in Room 219, and his wonderful insight. To Bob Gray, for his amazing tolerance for all the noise and random flying objects in Room 219 (much of which was my doing), for the enormous patience he showed when answering my innumerable, and often trivial, questions, for the quiet hilarity he added to the ongoing madness, for occasionally joining us when presented with the query, "Battle, Bob?", for jumping into the pool fully clothed, for being a huge source of support and inspiration, and for making a judgement call that took away a decision I could not make. To Guofei Jiang, for his interest in all that I did, for unwittingly being the source of hilarity and being good-natured about it, and for walking around The Green 20 times with me so we could determine GPS errors. To Katsuhiro Moizumi, for his hilarious, and occasionally intentional mangling of the English language, for the many cans of "whopis" he has opened in the lab, for his disarming frankness, for his simple approaches to answering my difficult questions, and

for his honest opinions. Many thanks to the people who joined the madness later, Michael Corr, Anush Kumar, and Jeanne Sucharitaves, for their encouragement, support, kindness, and friendship. Thanks must also go to Professors Ted Cooley, George Cybenko, Charles Sullivan, and Linda Wilson for teaching with a passion that made sure I learned something in every class I took. I am truly indebted to Luther Birdzell, Chrissy DeLorenzo, Todd Kerner, Satish Prabhakaran, and Kate Sherman, who all sat through ENGS 120 with me and made sure I finished my homework, studied for the tests, and learned a lot more than I could have on my own. Poring over intimidating equations in the Great Hall at strange hours in the night was made so much easier by their presence. A special thank you to Gustavo Mehas, for being the best roommate a person could have, for being my first friend at Dartmouth (and one of the best I've ever had), and for being one of the most interesting people I know. A heartfelt thank you to Gayathri Radhakrishnan, for being one of the most sweet and kind people I know and for being so caring; to Ravi Swaminathan, for providing me with the sanity checks I have always required, for sticking around for the past eight years and being willing to duke it out for a lot longer; to Vandana Vaidyanathan, for using that most fateful and for the record, inane signature, for being the irrepressible free spirit that she is, and for being forgiving of my many stupidities. Thanks also to Anita "Nitu" Chandrasekharan, Guruprasad "Guru" Giridharan, Sudarshan "Vicky" Chari, and Mahesh "unprintable" Shankar for hanging out with me for so many wonderful years. Many thanks to Niranjan "Bora" Bose, "Berther" Anush Kumar, Satish "Maams" Prabhakaran, Sriram "Koo" Srinivasan and Karthik "Koos" Vishwanath for the many happy hours of conversation, the infinite spicy hot meals, and perpetually open doors. Finally, I would like to thank my family. I cannot even hope of repaying my immeasurable debt to them. Appa, Amma, Mony, Gopi and people too numerous to mention have all provided me with an unquestioning support that I could not have done without. I haven't seen far, but what I *have* seen is only because I've stood on the shoulders of these giants.

There are, of course, the many, many people who should be in this section, but are not and to them I offer my most humble apologies. The only excuse I can offer is that I was constrained by the fact that I wanted to keep this section under two pages.

Contents

List of Illustrations

20. *Decision Point* Annotation Window

21. User-centric task tracing

22. Agent Tracker

Chapter 1

Introduction

1.1 Problem Statement:

The aim of Mission-flow Constructor (MfC) is to provide a workflow management system that facilitates the creation and instantiation of a dynamic, distributed workflow through a simple visual language that minimizes the amount of code written by a programmer.

1.2 Motivation

Business practice has come to signify many things in the recent past. In most cases, the term is defined as a set of procedures to follow in completing a transaction or making a strategic decision [WF94]. Business practice, with the above definition, finds a place not just in business environments, but in any form of large organization, particularly the military. Many strategic military missions can be modeled as a set of interrelated tasks, akin to procedures followed in the business world. With business process re-engineering becoming an important issue in the context of streamlining business practice it becomes necessary to evaluate and create tools that automate these processes, with special consideration being given to processes that are *ad hoc* and subject to run-time change. Mission-flow Constructor gets its name from the fact that this thesis was developed with the idea that it and subsequent incarnations would find use in military applications and hence, in this thesis, a distributed, dynamic workflow will simply be referred to as a "*mission*".

Most workflow management systems that are commercially available today are geared towards transaction processes in the business world [Zim98]. These workflows are traditionally static and well defined. In the real world, however, a mission rarely has a rigidly defined means of completion. In most cases, a mission is subject to run-time changes and disruptions. Human interaction, for example, could lead to exceptional conditions that lie outside those generated by usual computational processes. Transactional models do not adequately address these issues and it becomes necessary to develop a new model that provides the required functionality to execute a mission [Kou95]. A workflow management system based on such a model should be able to provide a completely general framework that can be adapted to very specific needs.

In this work, a mission is viewed as a completely generalized form of workflow that requires the workflow management system to adapt flexibly and dynamically to different schemata. This system models the mission as an interaction of distributed objects that contribute to the achievement of an end goal.

1.3 Problem description

Significant effort is required to develop code that executes a workflow across a distributed system, while conserving the hierarchical and temporal constraints implicit in it. This is compounded when one takes into consideration the fact that an organization will require many workflows with different schemata, guarantees functions. Principal issues of consideration are conservation of hierarchy, concurrency and synchronization [Sha97]. Other issues include efficient use of network resources, fault tolerance and ease of use. The execution of a mission therefore , has all of the above difficulties as well as the additional problem of being dynamic.

For example, consider a simplified version of the process of reviewing an application for admission to a graduate program at a university. First, there should exist a filtering procedure to determine whether the candidate has passed the minimum requirements, for example, a minimum undergraduate GPA and GRE score. If these minimum requirements are met, the application, along with supporting recommendations and transcripts, should be reviewed by various faculty members who independently evaluate the candidate. These independent opinions need to be collated and reviewed by the admissions coordinator who makes the final decision. (See Figure 1)

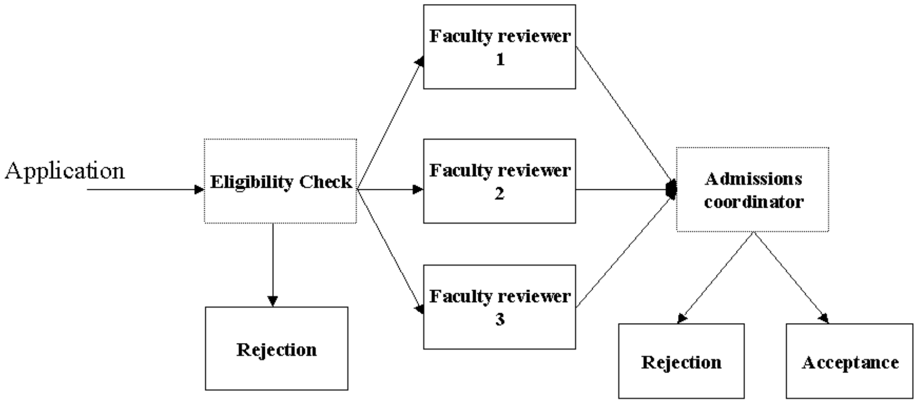


Figure 1 - Sample Workflow

This example brings to light some important considerations. The first of which is that this process can be mapped to two distinct workflows, whose topology is the same up to a certain point. First there is the case of the candidate not meeting the requirements and admission is then refused. Then there is the case where minimum requirements are met and other steps are to be undertaken. These cases can be collated into one workflow where a decision variable that chooses the next task is included. This new workflow is one whose topology or schema will change during execution. Notably, the first step determines whether or not the application is handed to the faculty members for review. The next step involves many people (the faculty reviewers) working on identical concurrent tasks. Once these concurrent tasks are complete, there is a need for synchronization (making sure that all reviews have been handed in) before the next task is initiated.

The above considerations - decisions, similar parallel tasks, synchronization points, etc - are among the most commonly found sub-graphs within a workflow topology. Coding these sub-graphs individually, even in the context of high-level languages is a repetitive task, and quite possibly a waste of time for a large organization that has a need to simultaneously deploy many such workflows. MfC eliminates a large portion of such repetitive coding by providing primitive constructs that encapsulate these topologies.

The backbone of any distributed systems is its effective use of network resources and ability to resist failure in the event that communication channels break down. Most distributed workflow applications assume that communication channels will always be open and that network failure does not occur. MfC incorporates some basic fault tolerance to network failure. Other sources of failure could be human error, absence, or unavailability.

Mission-flow Constructor (MfC) attempts to make transparent the distributed nature of these workflows by hiding the migration of and communication channels between the various tasks executing at different physical locations. However, the location of these tasks (and hence participants) is not hidden. MfC also takes away a significant amount of the coding required to generate agents that execute these workflows by providing a visual construction environment wherein concurrency, synchronization and hierarchical constraints are derived from the topology that the user provides by drawing the workflow on the canvas. Given that most workflow topologies consist of a limited number of primitive topologies, MfC makes simpler the task of drawing the workflow on the canvas by providing certain primitive constructs that encapsulate these commonly used topologies.

1.4 Overview

The remainder of this thesis is structured as follows. Chapter 2 provides background information on the topics that were of import to or resources for the development of MfC. Definitions of workflow terminology and some basic workflow theory are provided. Mobile agents and their suitability for this application are discussed and Mobile Agent Construction Environment (MACE), one of the earlier workflow management systems using mobile agents is dealt with in some detail. Chapter 3 lists the theoretical considerations of import to the building of a workflow management system while Chapter 4 details the implementation of Mission-flow Constructor (MfC), which is the body of work that this thesis supports. Chapter 5 collates the work into a few concluding remarks. Finally, Chapter 6 provides a few ideas and suggestions that could be put to use in creating future versions of this work.

Chapter 2

Background

2.1 Workflows

Workflows have gained acceptance as an excellent tool for process automation [Kob97]. The Workflow Management Coalition (WfMC) is a non-profit organization founded in 1993 whose mission is to "*expand the use of workflow by raising awareness, reducing risks and increasing investment value for workflows*." [WfMC95] The WfMC has published a reference model and has provided a set of standards for the definition, interoperability and execution of a workflow. The WfMC has also published a glossary of the standard set of workflow related terms. The WfMC model, like most commercial workflow products, centers around the theme of business process reengineering and transaction models of workflow enactment.

A workflow is defined by the Workflow Management Coalition *as "the computerized facilitation or automation of a business process, in whole or part* " [WfMC95]. A more usable definition, and one that will be used for the purposes of this thesis is , *"a sequencing of tasks that must be performed in order to accomplish a specific goal"* [Zim98]. Furthermore, a task will be defined as an activity to be performed by a single participant in the workflow [Zim98]. A participant in the workflow may also be referred to as a *workflow component*. Workflow management is defined to be "*the structured routing and tracking of information throughout an organizational process.* " [Kob97] A workflow management system (WfMS) is a tool that automates the execution of a workflow. The complete description of a workflow (one that encapsulates all the information required to execute it) is defined as the *workflow schema* or simply *schema*. A workflow schema is rarely linear, i.e., it is not always a simple sequence of a single task succeeded by another single task. There may exist whole sequences of tasks that are executed concurrently [Zim98].

Workflow topology is best understood through graph theory, where a graph consists of nodes connected by arcs. Traversal of a graph is effected by following the arcs from node to node. A traversal of an arc between two adjacent nodes is known as a *hop*. A graph where the arcs show explicit direction for traversal is a *directional graph*. Loops may occur within a graph where a particular path of traversal leads one back to the point from which traversal was initiated. Such a graph is known as a *cyclic graph* and the degree of the cycle is the number of hops taken to regain the initial position. For the purposes of this thesis, we will be concerned mainly with *directional a-cyclic graphs*. It should be noted at this point that this limits the kind of workflows that MfC can handle. However, it is anticipated that future versions of MfC will be able to manage workflows of more generic topology.

A workflow can be represented as a set of nodes connected by arcs. Each node represents a task and each arc provides scheduling information pertaining to the nodes that are connected by it. Each of these arcs must be directional in order to provide information regarding the temporal hierarchy. A graph is a visual map, and hence an excellent means of representing a workflow. Information about a workflow that can be obtained from its visual representation as a graph consists primarily of the temporal relationship between the tasks. It is important to realize that the functionality of the individual tasks is irrelevant as far as the topology of the workflow is concerned. While in some cases, the former may influence the latter, in general, the topology of the workflow can be completely described without any knowledge of the functionality of either the individual tasks or the workflow as a whole. Functional information about the workflow is rarely found in the visual representation and though it may exist, it is not always readily apparent. Functional information about a task is available from the task description, *i.e.* the code or instructions that specify what action is to be taken by that particular workflow component. Obviously, a graph will not contain all the information that a workflow is comprised of, just the visual topology. This topology, combined with the functional information of each task, provides the schema. Functional information would include the specification of workflow participants, the task to be performed, the format of the result, etc. This leads to an interesting conclusion, that a workflow needs to be specified at different levels, i.e., in more than one dimension.

2.2 Workflow Management Systems

A workflow management system (WfMS) is expected to fulfil two functions – process definition, which describes the workflow to be executed, and process execution, which is the enactment of the workflow. [CHRW98, Zim98, MLL97] provide a more detailed study of workflow enactment correctness and efficiency. Of particular interest in this work are the workflow enactment paradigms that are detailed therein, each of which are paraphrased briefly below.

Scheduler based: The workflow management system processes a schema and sends tasks or groups of tasks to various participants for execution. Many believe that these systems are ideally suited for well-defined, static workflows. Later in this text, it is explained why this model is well suited for a WfMS that deals with missions as collaborations between distributed objects. For the same reasons, MfC has been designed to come under this category of workflow management systems.

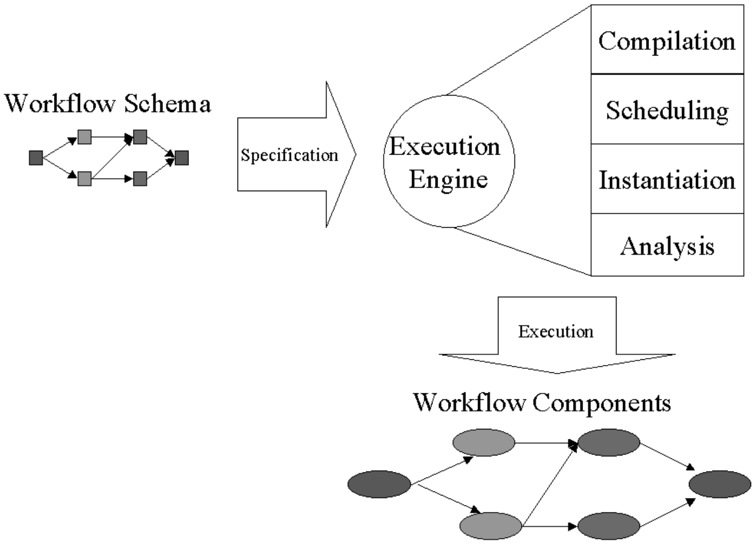


Figure 2 - Scheduler Based WfMS

Data-flow oriented: The workflow management system directs the workflow from participant to participant where the appropriate tasks are executed. In this case, partial specification of the workflow is acceptable, as the routing may be determined during the course of execution. MfC's predecessor MACE used an instantiation model that is similar to this enactment paradigm. MACE however, required complete workflow specification and did not provide dynamic routing capabilities.

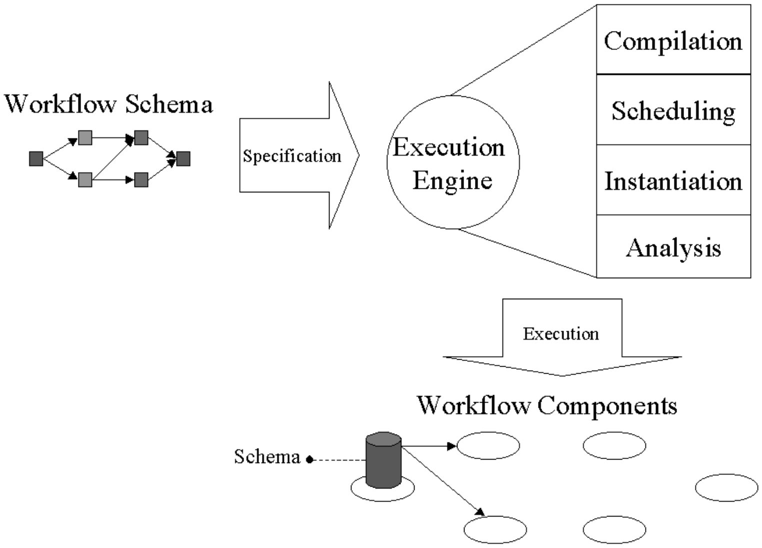


Figure 3 - Data-flow based WfMS

Information pull: In this case, the workflow specification itself is determined only after the workflow is instantiated and is usually created as a response to the need for information. This specification has been touted as being ideally suited for implementation with autonomous agents [CHRW98].

Workflow management systems have been standardized by a set of well-defined and meaningful terms and guidelines set forth by the Workflow Management Coalition (WfMC). The WfMC has also published a workflow reference model. (See Fig. 1)

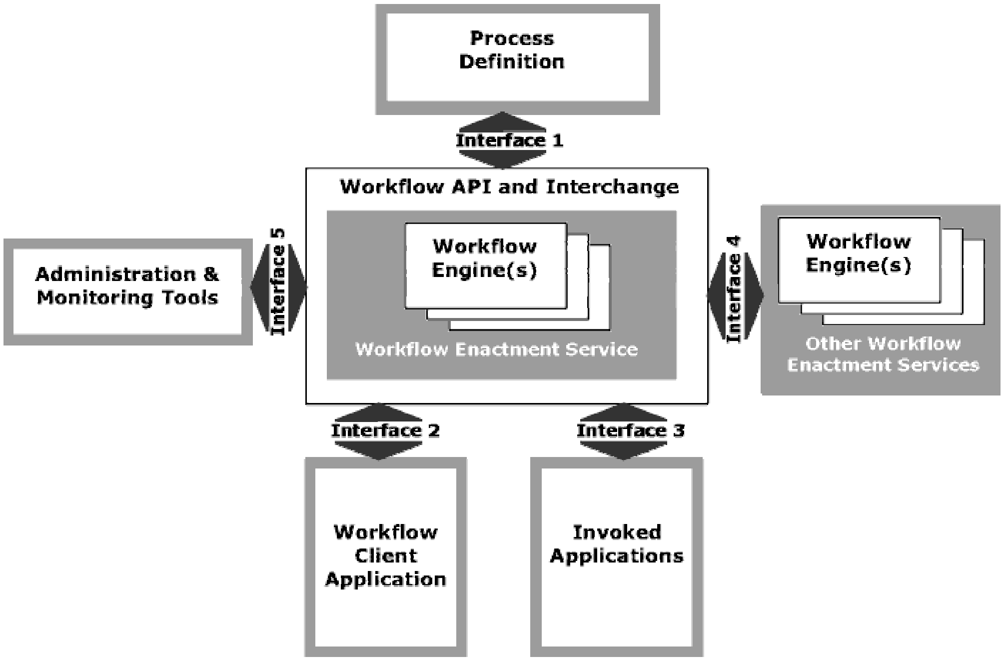


Figure 4 - Workflow Reference Model – from the Workflow Reference Model, Document Number TC00-1003, Issue 1.1, published by the Workflow Management Coalition. Used with permission.

Each of the interfaces shown in the reference model is called a Workflow Application Interface (WAPI). The WAPIs enable administration, monitoring, analysis, communication, integration with other applications, and semantically explain task functionality [CHRW98]. The various WAPIs are defined by the WfMC to provide true interoperability between all applications involved, if adhered to.

As pointed out in [Zim98], these standards are more geared toward business applications than generalized applications that are built as a group of interacting objects. Similarly, most commercially available workflow management systems and workflow solution software systems are geared primarily towards business and transactional models. Some workflow management systems available today are Mobile Agent Construction Environment (MACE), DartFlow from Dartmouth College, IBM's Flowmark, and Wang's OPEN/Workflow. DartFlow is a transaction based WfMS designed to be used over the Internet. DartFlow uses Java applets embedded in the user's web browser to generate a GUI and transportable agents to effect distributed workflow enactment [CGN96]. Flowmark provides a process definition facility for the specification and maintenance of process models. Also included is an interoperability standard (albeit different from the WfMC specification) to allow interfacing with other applications. The interoperability standard provides the user with the expected structure of information that passes from outside applications to Flowmark as well as that of information passed between member tasks [IBM]. Both DartFlow and Flowmark are limited in functionality because of the fact that they are transaction models of workflow execution. MACE, on the other hand, was a development environment for workflows, which also provided facilities for execution of the same [Sha97].

2.3 Mobile agents and the D'Agents system

A mobile agent is defined as a program that autonomously migrates from machine to machine in a heterogeneous network [Gra95]. By this, we mean that at any point, that the agent can suspend its execution, migrate to a different machine in the network with both its state and code, and resume execution from the point at which is suspended. Mobile agents offer a large number of advantages in the implementation of distributed applications, a few of which are detailed here. Since mobile agents are transportable, they allow local access to resources that are distributed through the network. Also, they are immune to network failure except when communication and migration across the network are to be undertaken. Mobile agents are most useful when one considers that development of distributed applications is eased by the fact that the communication channels between agents can be made transparent while the distributed nature, i.e. the location of the agents is not hidden. It is important that the distributed nature of an application is not hidden, as it is an inherent characteristic of the application that the user is aware of. Communication channels, however, are not an aspect that demands the users attention. Rather, the user is aware of the need for communication among the different distributed participants. Another important strength of mobile agents is their ability to react dynamically to a changing environment [Gra96]. Mobile agents find use in many applications such as e-commerce, adaptive active template management, workflow management, and network monitoring.

With regard to workflow management systems, mobile agents provide an efficient, robust and flexible means of implementation [CGN96, MLL97]. Agents can be delegated to perform the various tasks involved in the execution of the workflow. Since each agent can be made an independent program that carries the task specification with it, intermediate communication during execution is rendered unnecessary and concurrency of tasks can be exploited within the dictates of data dependencies.

Mobile agent technology has been under intensive research and quite a few mobile agent systems have been developed over the past few years. One such mobile agent system is D'Agents developed by Robert S. Gray at Dartmouth College [Gra97]. D'Agents is a flexible, secure mobile agent system that allows a developer to write mobile agents in high-level languages such as Tcl/Tk and Java. The D'Agents system that used Tcl/Tk was previously known as Agent-Tcl. D'Agents was selected as the agent system to be used for this project due to a number of reasons. Most important of all, MfC's predecessor, Mobile Agent Construction Environment (MACE), was built around the D'Agents system. Tcl/Tk is a high level scripting language, which makes it both portable and easy to learn. D'Agents being an in-house development of Dartmouth College, documentation and personal help were more easily available than with other agent systems.

D'Agents meets four main goals [Gra97]:

- Reduce migration to one command that may occur at arbitrary points. Capture of state information should be implicit.
- Provide transparent communication among agents
- Support multiple languages and transport mechanisms.
- Provide effective security in the uncertain world of the Internet.

D'Agents provides an agent server that keeps track of all agents running on its machine, accepts incoming agents, provides authentication, and routes agents to their appropriate interpreter. (See Fig. 2) The agent server also provides communication mechanisms for agents while also allowing direct connections between agents [Gra96]. D'Agents provides these services and mechanisms by adding a set of commands to the scripting language Tcl/Tk [Ous94, Wel95]. These commands include those required for an agent to migrate, communicate with other agents and register itself with local agent servers. Migration is achieved by capturing state, encrypting the state image and sending the state image with a digital signature to the agent server at the destination.

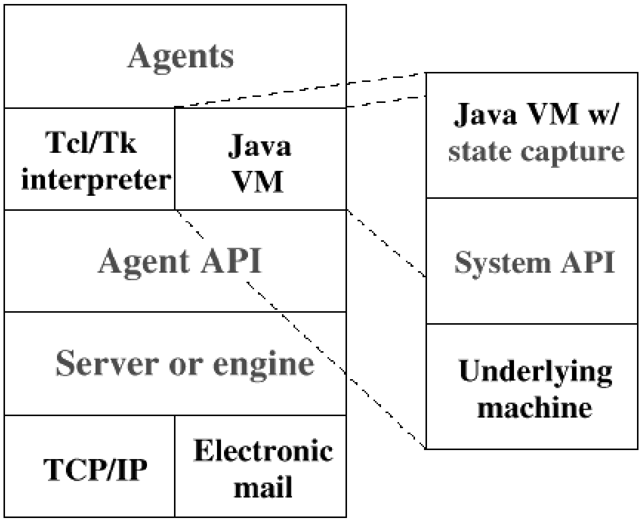


Figure 5 - D'Agents Architecture. This picture appears in [Gra97] and is used with permission

Agents generated by D'Agents are all uniquely identified (globally) by a four-field identifier. This identifier contains the symbolic name of the controlling server, the IP address of the controlling server, the symbolic name of the agent, and the numeric ID of the agent. The agent is assigned a numeric ID by the controlling server. The agent server ensures that no two agents have the same numeric ID or symbolic name. It is obvious that the agent identifier has information that has some redundancy. The utility of this redundancy will be seen later.

2.4 Mobile Agent Construction Environment (MACE)

Mobile Agent Construction Environment (MACE) was developed by Rohit Sharma as part of his Master’s thesis at Dartmouth College [Sha97]. MACE simplified the process of building mobile agents that were used to execute workflow by providing the user with a visual language to depict the workflow. The use of mobile agents was made transparent to the user without hiding the fact that the application was in fact, distributed.

As a workflow management system, MACE falls into the data-flow paradigm of workflow enactment. This is because MACE generates a single agent whose routing is determined by the dependencies of the individual tasks and the locations of the various workflow participants. The data-flow paradigm was described in [CHRW98] as the most suited to dynamic, goal oriented workflows. However, it is our contention that the implementation has some inherent limitations, which will be discussed shortly.

The implementation of MACE consists primarily of three components - the visual agent construction and monitoring environment, the compilation and execution engine and the critical path analysis module. For the purposes of this work, only the first two are of importance. MACE provides a graphical user interface (GUI) where the user can draw the workflow as a set of boxes (representing the various tasks) interconnected by arrows. (See Fig. 3) Each task is to be annotated by means of a set of descriptors that encapsulate the functionality of that task. The compilation engine then conducts a depth-first traversal of the graph representation to obtain the temporal hierarchy of the various tasks. The descriptors and code fragments that define the functionality are combined with the information obtained from the visual representation of the workflow to obtain the workflow schema. This schema is compiled to a D’Agents agent. Once execution is initiated, the agent follows the route established by the graph drawn by the user.

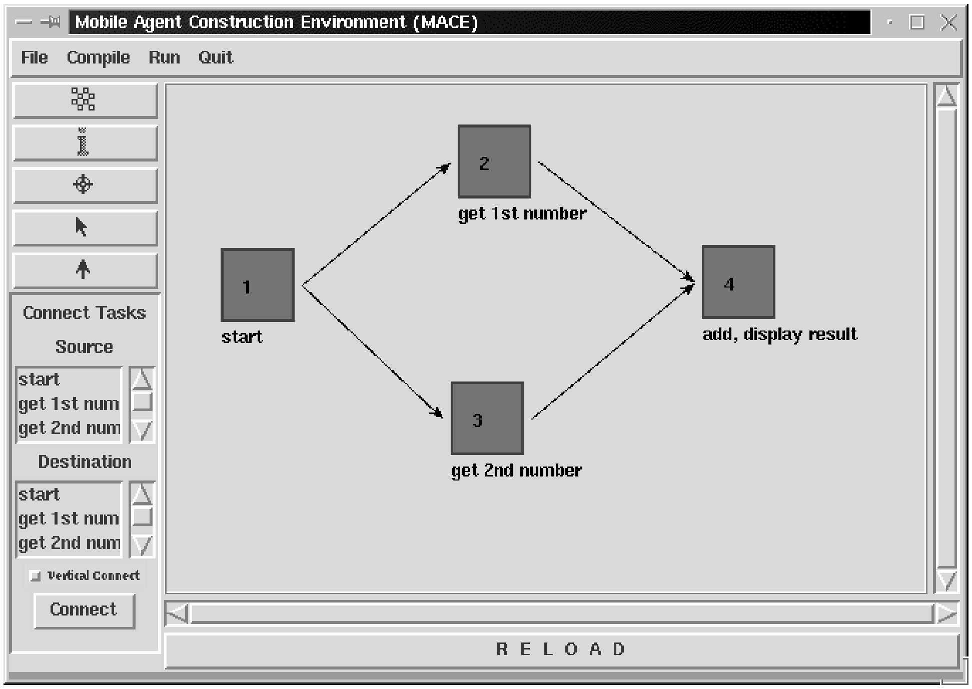


Figure 6 - MACE Screen Sample. Used with permission.

All MACE generated agents use only migration mechanism, namely ***agent_fork***. MACE generates a root task that spawns off the initial agents and serves as the monitoring agent for the workflow. Each task is implicitly assumed to execute on a different machine, so each task is mapped to an ***agent_fork*** command in the code generated by MACE [Sha97]. (See Section 3.5) Each of the tasks generated by the root agent spawn their succeeding tasks. Again, this is done by invoking the ***agent_fork*** command. Some important considerations arise from this method of effecting process migration:

- All the initial agents must carry the code required to execute their succeeding tasks. This is necessary as the ***agent_fork*** command creates an exact copy of the agent that invokes the command. This could lead to scalability problems when extremely large and complex workflows are to be enacted.
- All agents carry the complete workflow schema. This is an example of strong migration, where the entire workflow is available at every node of execution. While strong migration is desirable in many cases, in this case, agents executing later in the timeline of the workflow

are carrying what might be a large volume of completely code that will not be executed. Again, this could lead to scalability issues.

- An agent that has forked the new task to its required destination must terminate itself. Otherwise, there will exist two agents that are executing the exact same task, one of which (the parent agent) should not exist.

The last point listed above was adequately addressed in MACE, but the first two considerations were deemed to be inescapable prices that were to be paid in return for being able to use only one migration mechanism. With scalability being an issue of consideration in later versions, it became necessary to re-evaluate the migration mechanisms that were earlier deemed acceptable.

Some elementary monitoring capability is also provided by MACE. During run-time, the user can monitor the progress of the workflow by means of updates that are provided by the monitoring system embedded in MACE. Messages are sent to the root agent upon completion of each task. The GUI is then updated by darkening the boxes representing the tasks that have completed.

One of the important drawbacks of MACE is the fact that it does not respond adequately to a task that fails. Once an agent has failed (for whatever reason), if any other agents are awaiting results from the previous agent, neither the monitoring service nor the agent that died inform the remaining agents. This results in “hanging agents.” A hanging agent is one that is caught in an event loop or otherwise awaiting the occurrence of an event that neither can nor will occur. A hanging agent is usually terminated by the agent server on the local machine. An agent server usually imposes a predefined limit on how long an agent may execute on the local machine. Once exceeded, the agent server forces the termination of the agent in question. MACE by itself does not prevent the occurrence of these hanging agents and in the event that agents are left hanging, MACE does not force their termination. A hanging agent represents an unacceptable state of execution/termination for the workflow.

To conclude, MACE was an easy to use tool that put mobile agents to work in enacting a distributed workflow. MACE provided a very high level of abstraction in the process of creating mobile agents to the extent that MACE was able to hide the fact that agents were being used. A visual language was proved an excellent means of reducing the time and effort required to describe a distributed workflow [Sha97]. As in the case of most prototypes, MACE suffered from a variety of deficiencies, some serious. MfC attempts to amend some of these drawbacks, while also breaking ground in areas not covered by MACE.

Chapter 3

Design Considerations

In this chapter, we briefly describe some the questions that arise when we consider the implementation of a distributed, dynamic, workflow management system. Foremost we must consider the requirements of such a system. These are enumerated and discussed below.

3.1 Requirements

Many texts have been written on the subject of requirements of a WfMS and the services it should provide. This discussion is aimed primarily at distributed, dynamic workflows, and hence this section collates those requirements deemed relevant. A broader approach to these topics can be found in [CHRW98, Kob97, Kou95, MLL97, MN, Zim98].

3.1.1 Distributed participants: The system should support workflow components and participants that are separated geographically. This means computers and other (electronic) resources distributed throughout a network as well as people in different regions. Thus, the system must account for the uncertainties that accompany such distribution. These uncertainties include network failure/downtime, unavailability of people, and computer failure.

3.1.2 Dynamic schemata: The system must allow changes to the schema of the workflow being executed without causing the workflow to go into an unacceptable state of execution or termination. Dynamic changes to the schema could mean the inclusion of new participants, exclusion of some participants, modifications to the participating objects, or replacement of participants. This is far different from traditional workflows, which are characterized by their static schemata. Implementation of dynamic systems requires a significantly different approach. Dynamic sequencing or a change in topology is another aspect of changes to the schema. The WfMS should allow changes to the sequencing of the tasks even after the workflow has been instantiated.

3.1.3 Complex Schemata: It is necessary that the WfMS be able to handle workflows whose schemata are neither linear nor simple in their topologies. Even through the use of a GUI, specification of complex workflow schemata is not easy. The WfMS must provide means of simplifying the specification of a complex workflow. Supporting the execution of such complex schemata is equally critical. Execution of complex workflows carries with it certain difficulties such as task concurrency, data consistency, and efficiency and effectiveness of monitoring.

3.1.4 Scalability: With workflow technology being applied in almost all spheres of process automation, a WfMS will find application within a small workgroup as well as a large enterprise. A WfMS should be able to handle large workflows regardless of the complexity of the topology.

3.1.5 Concurrency of workflows: It is desirable for a WfMS to support the concurrent execution of multiple workflows of a given schema, *i.e.*, multiple instances of the same workflow should be supported. While this could be accomplished by setting up an instance of the WfMS for each of the jobs being processed, such an approach would lead to problems when different instances of the WfMS (all of the same authority) requested the services of the same workflow component. It is necessary to develop intelligent criteria that help a WfMS schedule the usage of the various workflow components by the different instances of the workflow being executed.

3.1.6 Monitoring: A WfMS should be able to provide the user with status information on all the tasks associated with the complete workflow. Monitoring should include the means to log an execution history or audit trail. This generates an information base that would be useful for security purposes [CHRW98]. The tracking mechanism should be able efficiently monitor the execution state of every task, as well as input and output data generated by a large workflow.

3.1.7 Reliability: A WfMS must guarantee the correct execution of a workflow in each instantiation. In most cases, this would simply mean the guaranteed execution of all tasks and the achievement of the final objective. However, in the case of a *mission*, (a distributed, dynamic workflow) neither of these can be guaranteed due the nature of the environment in which it executes. A more applicable set of guarantees for the reliability of a WfMS would include contingency plans in the event of task failure, communication breakdown, or human absence. Alternatively, a WfMS should be able to guarantee that execution of a workflow ends in one of many *acceptable states of termination*. Acceptable states of termination should be predefined and should include the status of goal satisfaction. A WfMS should also be able reject a workflow that cannot meet the guarantees or is simply infeasible [CHRW98].

3.1.8 Failure atomicity and recoverability: Failure atomicity is one of the most desirable properties of a WfMS. An excellent example for failure atomicity is a bartering workflow. There are two tasks involved here: giving the other party your item and receiving the item that you want. It is necessary that both of these tasks be completed for the trade to be successful. In this workflow, it is imperative in that either all or none of the tasks complete successfully. It would hardly be considered a trade if one was simply to give away possessions. In other words, "*a workflow should execute entirely, or not at all*." [CHRW98] Since failure of workflow components is an inevitability, we can only achieve failure atomicity by guaranteeing the ability to "undo" the tasks that have already completed. This brings us to the topic of recoverability. Recoverability falls into two categories: rollback, or backward recoverability, and resuming execution from a state image, or forward recoverability. Rollback assumes the ability to undo any and all actions taken by each task. Rollback is not always possible in the computing world and even less so in the administrative world. For this reason, backward recoverability is rarely implemented in a WfMS. In the context of implementation using mobile agents, forward recoverability is the more viable option and is made easier when there is strong migration of tasks [CHRW98].

3.1.9 Interoperability: Workflow interoperability is of two types: specification interoperability and execution interoperability. Specification interoperability guarantees that workflows specified in other systems can be processed. Execution interoperability guarantees the co-operation between different systems. Both require a set of standards governing the interface between a workflow schema and a WfMS. While the WfMC has provided some interface specifications, for a variety of reasons, almost none of the commercially available WfMS packages adhere to this standard [WfMC00]. Interoperability is one of the most difficult guarantees to implement.

3.1.10 Flexibility: A WfMS should not limit the user by the type of functionality available, specification method used, or execution environment. [CHRW98] treats the WfMS as nothing but an execution environment, in which case it is possible to make both specification method and language open to the choice of the user without compromising the functional capabilities of the WfMS. Since most workflow management systems offer a development or workflow specification standard in addition to execution capabilities, a large portion of the potential flexibility of these systems remain unrealized.

3.1.11 Security: Security requirements encompass a wide area with respect to a WfMS. There is first the question of authority. Within an organization, it is necessary to ensure that creating an instance of a workflow is done only by a user of such authority to do so. Modification of a workflow during execution should also require verification of authority. The question of authentication also arises. A workflow component should be able to verify the identity of the components that send it data/messages. Also, data in transit should be protected by means of encryption.

3.2 Mobile Agents in Workflows

Traditional approaches to implementing workflows using mobile agents involve the creation of an agent that carries with it the complete workflow schema [MLL, Sha97, Zim98]. This agent migrates (in sequence) to the necessary machines to execute the various tasks. Once all tasks have been completed, the agent migrates to the “home” machine and provides the user with the results. A WfMS that uses the single agent approach hence makes use of the data flow enactment paradigm described in **Section 2.2**. This approach uses the most obvious capability of a mobile agent – migration. The understanding that a mobile agent may act as a personal “agent” (in the human sense of the word) for a person or an application also contributes to that fact that this approach is the one most widely used. This implementation has distinct advantages such as strong migration, ability to schedule dynamically, reduction of human interaction, etc. However, we contend that the single-agent approach is not ideally suited to the implementation of distributed dynamic workflows, and that the advantages of the single-agent approach can be achieved through other means. If a single agent is to execute the entire workflow, concurrency of tasks cannot be exploited – tasks must be scheduled in a linear sequence. MACE uses a modified version of the single agent approach and solves this problem by allowing the workflow agent to create sub-agents that execute concurrent tasks.

One of the major issues that arises with the use of the single agent approach is scalability. The agent that executes the workflow must carry with it the entire workflow schema. With a large and complex workflow, this agent is bound to be of prodigious code size. This defeats one of the primary advantages of using mobile agents – reduction in network traffic. Each time the workflow agent migrates, it carries the information required to execute subsequent tasks as well as that required for preceding tasks. Once a task has completed, its code becomes unnecessary. With the completion of each task, the percentage of useless and unnecessary code that the agent carries increases. Consider the case of a linear workflow consisting of ten tasks of equal code size. By the time the workflow agent executes the migration to the location where the final task must execute, 90% of the code the agent carries has been rendered useless. In linear workflows, this percentage increases linearly (as the ratio of tasks completed to the total number of tasks) with migration, provided all tasks are of equal code size. With workflows of complex topologies, the percentage of useless code carried by the agent increases much faster as it completes the schedule. Rigorous mathematical models of these situations are beyond the scope of this work.

An implicit and often unstated characteristic of workflows is the functional independence of tasks. While functionality can depend on the *result* of other tasks, there is no dependence on the functionality of other tasks (there exist only data dependencies). Traditional execution models that use the single-agent approach ignore this fact by encapsulating the functionality of the entire workflow within one agent. While this does not create functional dependencies, it does not allow distribution of the independent objects.

We propose to abandon the single-agent approach and use many agents, each with limited functionality, to execute the workflow. This leads to the question of how many agents are necessary. One solution is to use as many agents as there are tasks. We assume here (both MACE and MfC are built using this model) that tasks are coded by the user and that the WfMS provides a wrapper that enables execution, communication, and migration. In the case that many tasks are to execute at the same location, each requires an individual wrapper. We contend that the code used for wrappers can be reduced by collating the functionality of tasks based on their location, *i.e.*, using as many agents as there are locations. It should be understood that within this argument, “location” and “workflow component” are synonymous. With this synonymy in mind, one begins to see the importance of the association of a task (functionality) with its workflow component (user or location). We contend that this is in fact the most important association for a WfMS that uses mobile agents. This association not only enables reduction of the size of agents, but also provides an excellent resource for monitoring the efficiency of execution of a workflow. Knowledge of the location of a task (and hence the agent executing it) also provides the backbone for communication and enables transparent communication with the various agents.

Another important question that arises when using mobile agents is that of deciding when an agent should migrate. As stated before, traditional implementations make utmost use of the ability of an agent to migrate. Many texts discuss the utility of migrating process when implementing a mission. Frequent migration, however, makes a mission more susceptible to failure due to network uncertainties.

Also previously discussed was the waste of bandwidth that accompanies frequent migration. The functional independence of tasks leads one to the conclusion that passing results between tasks is the only communication that is necessary for successful completion of the workflow. This statement would be true in the context of static workflows and completely reliable network situations. When we come to the concept of a mission, information regarding the dynamic changes of the workflow schema is also required. It should be noted that results from previous tasks are still the only information required by a task for its (not the entire mission's) successful completion. Hence, we believe that message passing (short messages) can be more efficient in terms of network resources than process migration. Process migration, however, is necessary to enable distributed, platform independent workflow execution. Migration mechanisms available in D'Agents are discussed in **Section 3.5**.

Thus, we are led to the conclusion that the best means of implementing a mission using mobile agents is to use the scheduler-based model discussed in **Section 2.2**. Here, a workflow schema is submitted to the execution engine, which then sends tasks to the appropriate workflow components. In the model we have implemented, processes migrate only once and that too only to provide an instance of a workflow component that is required at a location. These components are activated by the various events (usually task completion) that occur during the execution of the mission. Once the workflow component completes its task, the component terminates itself. This model is very similar to the many distributed objects models that have been discussed and implemented (as prototypes) [CHRW98, Kou95, Zim98]. A comparison of the two models yields a few differences in the semantics involved, but the concepts driving them are virtually identical.

3.3 Primitive Constructs in Workflow Specification

Previously discussed was the fact that almost all workflow topologies consist of a limited number of sub-graphs. In this section, we discuss the sub-graphs that are most commonly found in workflow topology and describe possible implementation considerations. Considering that a workflow topology is rarely linear, we immediately note that there can exist multiple concurrent tasks. This would imply that there might exist in a topology a "*split point*", where a single task provides the input for or initiates more than one subsequent task. Conversely, there could also exist a "*join point*", where a number of concurrent jobs must together provide input or initialization data for a single task. These sub-graphs can be generalized as *n-destination split points* and *n-source join points*. These generalizations serve only the purpose of encapsulating a commonly used topology, not functionality.

At this juncture, it is important to note that primitive constructs for workflow specification can be of two types – topological primitives and functional primitives. The advantage of using topological primitives in describing workflows is that the time taken to draw a workflow is reduced. However, tasks must still be individually annotated with functional information. With functional primitives, commonly used functionality is encapsulated and may be reused as and when necessary within a given topology. Here, functional specification of a workflow is made easier but not the topological representation. Independently used, these two types of primitives cannot alleviate much of the workload associated with complex workflow specification. Here, one can draw the conclusion that, more than using primitives that are either strictly topological or functional, some form of hybrid primitives that take the form of one while enforcing some constraints on the other would be useful.

In many workflows, the topology of a workflow imposes some constraints on the functionality of tasks. Notably, some topological sub-graphs can indicate similar, repetitive, or decision-making functionality of the tasks contained in them. For example, most often, the concurrent tasks that succeed a split point are of the same functionality. In the case of the admissions review example we presented in Section 1.3, the application for admission is handed simultaneously to three faculty members who independently review it. (See Figure 8) Many such examples can be thought of, wherein independent opinions are to be obtained or more generally, the same data is to be processed in the same way by different participants (usually resulting in different results).

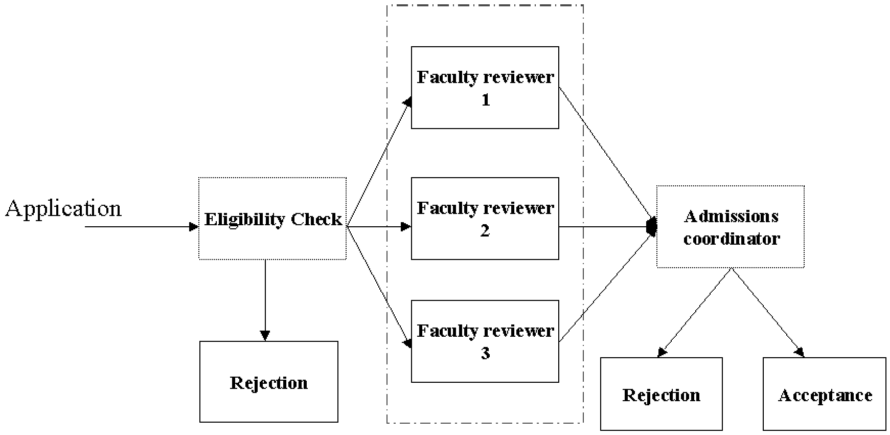


Figure 7 - Similar concurrent tasks - the "*scatter*" primitive

Considering that such sub-schemata within a workflow are quite common, we propose a primitive construct to be called "*scatter*" that encapsulates the following characteristics.

- The *scatter* construct renders the preceding task an *n-destination split point*.
- All concurrent tasks that are successors of the split point are of the same functionality.

While the join point seems to be the exact converse of the *scatter* primitive, there exist many significant differences. The *scatter* primitive gets its name from the fact that it literally scatters processes. The join point is more a synchronization point than a process node. (It should be noted that the join point is a synchronization point only in the finish-to-start execution model that has been implemented.) Under that consideration, it is difficult to imagine tasks "joining". Rather, the information that these tasks generate, i.e., their result data can be collated or joined. Here we propose a "*gather*" primitive that serves as a synchronization and data collation point in the workflow. It should be noted here, as will be seen in

Chapter 4, that the *gather* primitive does not provide functionality that does not already exist in MACE or MfC. Rather, the *gather* primitive is provided for the sake of completeness and more importantly, to showcase an important primitive commonly found in workflow schemata. The join point in our admissions review example would be the point at which the various faculty reviewers handed in their opinions. It should be noted here that the functionality of the task that represents the join point is not of any consequence to what the primitive provides. The gather primitive should not be considered a direct converse of the scatter primitive for the simple reason that the *scatter* primitive scatters processes, while the *gather* primitive gathers data. While it is possible to scatter or disseminate information to many tasks, doing so does not ease the process of workflow specification. In a workflow, scattering information would simply be the sending of result data to succeeding tasks. This is quite easily implemented and is, in fact, the way low-level workflow specification is done. Gathering tasks is clearly not possible.

Another form of a primitive construct that is commonly found in workflow schemata is the *decision point*. The *decision point* is a task node that has multiple succeeding tasks, a subset of which are to be instantiated. The decision of which tasks are to be initiated is made using previously defined criteria that are evaluated at run-time. Looking back again at our example, we see two decision points. The first is the point at which the candidate's eligibility for admission is reviewed. The second is when the admissions coordinator makes a decision as to whether or not the candidate should be admitted. (See Figure 8) There exist differences between the two decision points, which will be used to arrive at how the primitive construct is to be defined. The first decision point has functionality that can be automated while the second requires human intervention. Also, the first decision point has four succeeding tasks, but only two decision states while the second has an equal number of decision states and succeeding tasks (a one to one mapping). These differences lead to two important conclusions, first of which is that a decision point must have open functionality. In the specific case of MfC, we do not impose any restrictions on the code (written by the user) that represents the functionality of the decision point. The second conclusion is that there need not be a number agreement between the number of succeeding task to a decision point and the number of decision states that it can take. For instance, a task may have larger number of tasks than decision states as seen in our admission review example. The converse is also true, i.e., many different decision states can be mapped to a smaller number of tasks. In addition, multiple decision states can be mapped to the same task and vice versa. This can lead to a large number of parameters that need to be specified in order to adequately describe a decision point.

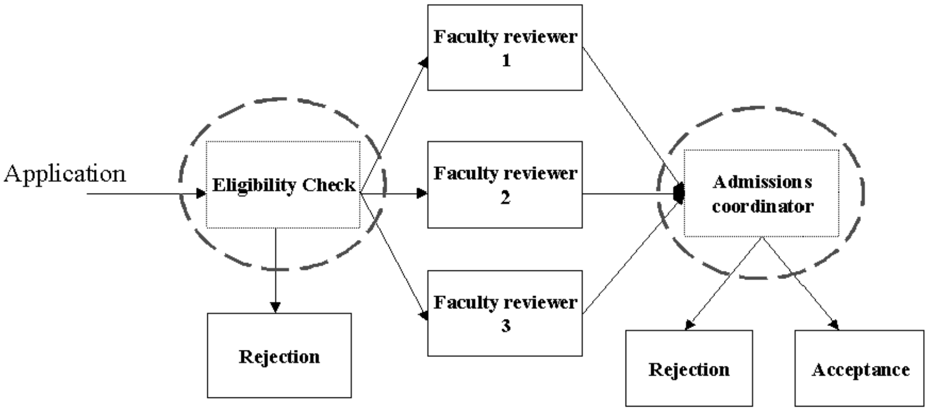


Figure 8 - Decision points in the sample workflow

For the purposes of this thesis, the *decision point* primitive will be defined as an *n-destination split point* with conditional execution of the successors. Considering the number of parameters that need to be specified in order to define the decision point, we have implemented a simplified *decision point* that imposes the following restrictions. The user must ensure that the decision variable is set to the appropriate state. This is done by programming either for human interaction or for a computational result. The user must also specify the mapping of decision states to task instantiation.

The final primitive that we propose has been termed the *sentinel* node. In many, many cases, we find the need for tasks that must execute repeatedly until the workflow has completed execution. An example of such a case is a weather monitor. For as long as say, a weather forecast workflow is executing, there may be a need to monitor current weather conditions. In that case, the weather monitoring task node would have to constantly execute until the weather forecast workflow has completed. There are many such examples of monitoring or *information-push* tasks. These tasks by themselves are *single-degree cyclic graphs*. Implementation of cyclic graph structures is outside the scope of this work. However, as a starting point, we have considered and implemented a *sentinel* with the following characteristics. The *sentinel* executes in response to a request. Each time a *sentinel* is given an information request, it executes the code that defines its functionality and returns the result data. The *sentinel* remains in a “wait mode” between information requests and until the workflow completes execution. One important consideration for a *sentinel* is to ensure that requests are handled in sequence and not concurrently in order to avoid data hazards. A better and more involved implementation would require that the *sentinel* execute repeatedly and without interruption, posting results in real time. These results can be time stamped and made available to workflow participants that request them.

Of course a specifying workflow using only these primitives would require far more effort than using a low-level, first-pass specification method. So the generic task node has also been made available. The generic task node can have any number of preceding tasks, any number of succeeding tasks and is of open functionality. To recapitulate, below is a list of the primitive constructs that have been proposed and implemented in MfC.

- *Scatter*: Allows the user to define any number of similar concurrent tasks as one object. All tasks are of the same functionality and take the same input(s).
- *Gather*: Collates data from previous tasks.
- *Decision point*: A task node that imposes conditional execution of succeeding tasks.
- *Sentinel*: Executes each time an information request is received.

3.4 Visual languages

In order to facilitate the communication of complex mission schemata between the user and the WfMS, there needs to be a specification standard that is easy to understand. The first specification mechanism that comes to mind is a one-dimensional method, which involves a complete, almost textual, description of the schema. This would involve detailed listings of task functionality, locations, participants, etc. While one-dimensional or single pass methods of workflow specification do exist, they are far from optimal [Zim98]. A single-pass workflow specification is tedious, inefficient and is impractical for large workflows. With distribution and dynamism as added factors, even small workflows become unwieldy in terms of their specification. Since most one-dimensional specifications are text based, quickly parsing and understanding such descriptions is difficult.

Graphical user interfaces (GUIs) make such communication easy, understandable and more productive. While a GUI provides an easy communication medium between the user and the WfMS, it does not necessarily provide the user with easy method of specifying the workflow. Better specification methods would involve a more high level specification that allows the use of complex constructs modeled as primitive constructs. "*Goto*"-style control flow should be avoided in such high-level specification methods [Zim98]. It must be noted that specification of a workflow involves not only the topology of the workflow, but also the specification of the individual tasks in terms of their inputs, outputs and functionality. It becomes imperative to use a method that allows specification of a workflow at more than one level. Such methods are best implemented as visual languages.

A visual language is a means of constructing a complex image from a set of simpler images where the result has a meaning distinct from the parts that comprise it [GBCK94]. More simply, a visual language is a programming system that uses a pictorial notation and extracts semantic information from it. Most visual languages require more than a one-dimensional approach to specification. In those cases, the pictorial notation is the first dimension of specification after which some textual annotation will be required. One of the most compelling arguments for the use of visual languages in any form of application programming is the fact that humans process pictures faster and easier than text [Naj94]. In the case of workflow specification, visual aids are of paramount importance when one considers that the most common representation of workflows is visual.

Most visual languages can be classified as either control-flow or data-flow based systems. Control-flow systems are a pictorial depiction of control flow (usually in the form of flowcharts) and do not entirely eliminate "*goto*"-style statements. Data-flow based visual languages rely more on a workflow-style of programming wherein image constructs represent procedures or objects and their inter-connection denotes data flow. It seems obvious that a data-flow based visual language would be ideal to specify a workflow. MACE provides an excellent example of a visual language for workflow specification. It should be noted that MACE provides the user with both a visual programming environment as well as a program visualization system [Sha97]. In view of this, many aspects of the MACE GUI have been ported to MfC.

3.5 Migration Mechanisms

D'Agents provides three mechanisms for agent migration. All three use a single command to effect migration and can be invoked at arbitrary points in execution. A detailed explanation is available in [Gra95], however, a brief outline of these mechanisms is given below.

- ***agent_submit**:* This migration mechanism takes as one of its arguments a Tcl/Tk script. This script is submitted to the agent server at the destination as a new agent. The script is executed when the new agent registers itself with the agent server at the destination. This command can be thought of as the command used to spawn or create a new agent (a child of the agent that submitted it). (See Figure 9)

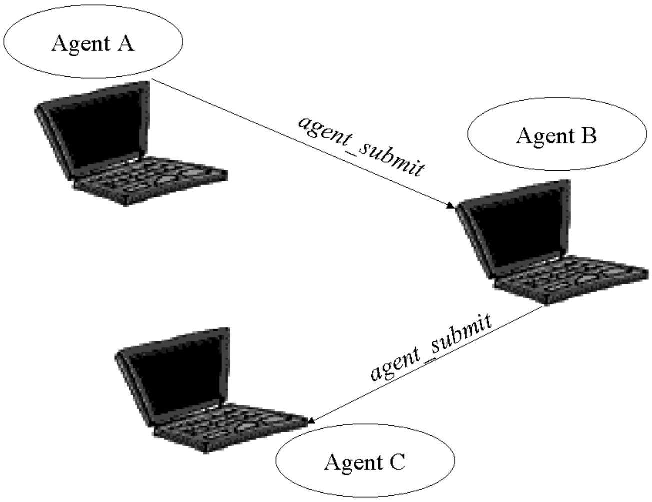


Figure 9 - *agent_submit*

- ***agent_jump**:* When invoked, this command captures the internal state of the agent, and transmits the state image to the destination server. This server then recreates the state of the agent and allows the agent to resume execution. (See Figure 10)

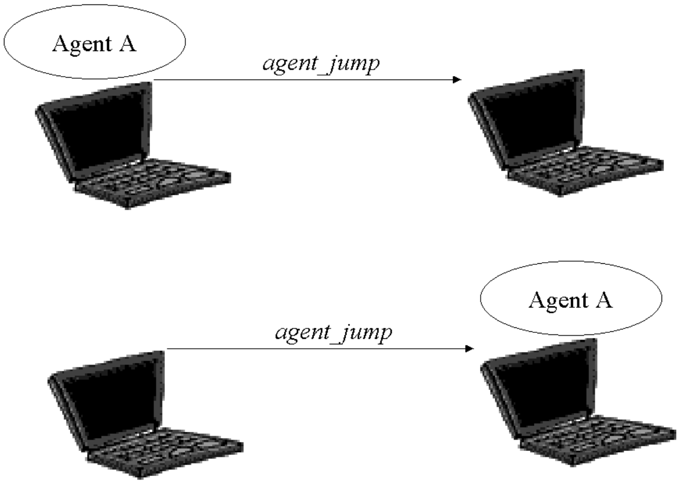


Figure 10 - agent_jump

- ***agent_fork**:* This command is analogous to the Unix fork command. It submits an exact copy of the agent that invoked the ***agent_fork*** command to the destination specified. Both parent and child agents then resume execution from the point at which the fork was initiated. (See Figure 11)

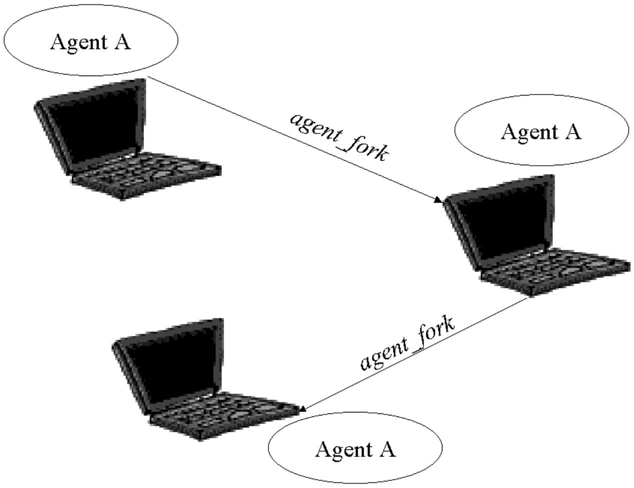


Figure 11 - *agent_fork*

The ***agent_fork*** command was the sole migration mechanism used in MACE [Sha97]. Section 2.4 enumerated the various drawbacks associated with the use of the ***agent_fork*** command. The ***agent_jump*** command suffers from similar setbacks. If the various agents we submit are to jump from location to location, different implementations can be used. The first, of course, is the single-agent paradigm, which we have decided to abandon for reasons discussed previously. For the sake of completeness, this implementation in D'Agents will also be considered. If a single agent is to be used, then concurrent tasks cannot be executed concurrently. To enable concurrent processes, "child" agents must be created and other migration mechanisms such as *submit* or *fork* must be used. Another possible implementation is to use a migrate-once mechanism, create all agents at a controlling location (where the WfMS is running), and have the various agents jump to the desired locations. This implementation requires that the agents be created at the location of the WfMS. This could be done either by generating D'Agents scripts (containing appropriate ***agent_jump*** commands) that are executed by the WfMS or by submitting agents to the location of the WfMS and having the agents jump from there to the necessary locations. The first of these methods requires the generation of a stand-alone D'Agents script that must be written to disk, made executable, and called by the WfMS. The second implementation uses the

agent_submit command. The ***agent_submit*** command, however, is ideally suited for this application. When multiple agents are to be used, each agent may be directly submitted to the location at which it must execute. This provides us with the single migration mechanism that is efficient and simplifies implementation. It should be noted that there is no compulsion that a WfMS (that uses mobile agents) should use only one migration mechanism. Rather, this is done to simplify implementation. Ideally, on a case-by-case basis, the WfMS should be able to decide which migration mechanism to use to create an instance of an object. This would require the development of intelligent criteria that force such decisions as well as an in-depth look at the workflow schema before execution.

3.6 Communication Mechanisms

With any distributed computing application, communication between the distributed objects is necessary. Dependent on the application is the content of such communication. In this section we deal with those requirements necessary for a dynamic, distributed WfMS. Issues such as type of communication, choice of mechanisms, and content are addressed. Since missions are assumed to be running on different hardware platforms, it is critical that both low-level and high-level considerations are addressed. Low-level concerns include choice of communication protocol and hardware dependencies. Low-level concerns in MfC are addressed by the D’Agents system and only a brief description is provided below. High-level considerations center around the transfer of the semantic content of the messages. In the context of high-level considerations, we discuss the type of messages expected and appropriate responses. High-level considerations obviously affect the interoperability of various systems, but we will restrict our discussion to the use of one WfMS and in particular to MfC.

D’Agents provides communication mechanisms that allow inter-agent messaging as well as the capability for agents to open direct communication channels amongst themselves. Messages are passed between agents using the ***agent_send*** and ***agent_event*** commands, for which corresponding commands to receiving those messages are also provided. A direct connection between agents can be established using the ***agent_meet*** command. D'Agents allows agents to communicate amongst themselves using any of these mechanisms, each of which are detailed below. A more in-depth discussion is available in [Gra97].

Message passing: The message-passing model of agent communication involves two primitives – **send**, which sends a message to the intended recipient and **receive**, which enables the receipt of a message. Message passing leaves the developer with the responsibility of deciding appropriate responses to the various messages, obtaining addresses of recipients and handling exceptions that could arise [Gra97]. D'Agents provides two mechanisms for message passing – ***agent_send***/***agent_receive*** and ***agent_event***/***agent_getevent***.

- ***agent_send/agent_receive:*** The ***agent_send*** mechanism sends a message consisting of a numeric code and a string, both to be provided by the programmer. The message is received using the ***agent_receive*** command, where the programmer specifies two variable names one of which is set to the numeric code received and the other to the message string.

- ***agent_event/agent_getevent:*** The ***agent_event*** command is almost exactly like the ***agent_send*** command and differs only in that the message sent consists of a tag and string. The difference here lies in the fact that a tag is not limited to being numeric. With respect to these similarities, later versions of the D'Agents system will have only the ***agent_event*** command.

Meetings: The D'Agents system allows a more direct and bandwidth-efficient means of communication among agents, namely meetings. Meetings between agents are established using the ***agent_meet*** command. The agent_meet command is a request for a meeting. Meetings can be accepted using the ***agent_accept*** command or rejected using the ***agent_reject*** command. Once a meeting is accepted, the controlling servers establish a direct TCP/IP connection between the two agent processes. Once such a connection is established, agents may read from or write to the socket opened, using commands that are provided in D'Agents. It should be noted here that at least two messages (***agent_meet*** and ***agent_accept***) must be passed before a meeting can be instantiated. Hence, a meeting can be more efficient than message passing only if the bulk of data is substantially higher than the overhead generated by the two "handshake" messages.

D'Agents allows the programmer to automate the receipt and response to messages, but not meetings. Meeting requests can be handled automatically, but not the content of the meeting. D'Agents uses an event-driven programming paradigm to enable such automation. The D'Agents system is designed with the intent of making message passing the preferred means of communication among agents (for transfer of semantic content). Meetings are to be used for bulk data transfer. A *mask* can be added to an agent's code to allow it to automatically handle various messages. A mask is an event handler for the various messages that may be received. Masks can be added to either or both of the message-passing mechanisms and thus specify which event handlers respond to the different message types. An important point to note here is that whenever a D'Agents agent encounters an error, the controlling server sends a standard exception to the agent's parent using the ***agent_send*** mechanism. In view of this, we have reserved the ***agent_send*** command to transmit error messages and the ***agent_event*** command for routine communication. Also, the ***agent_send*** command is limited by the fact that apart from the message string, additional information can only be furnished in the form of a numeric code. Future plans for MfC include use of the ***agent_meet*** construct for the transfer of code to allow changes in functionality during the course of execution of a workflow.

Chapter 4

Implementation

Mission-flow Constructor (MfC) is implemented as a single executable D'Agents script. When the MfC script is executed, it registers itself with the agent server on the machine on which it is running. The MfC script itself is thus an agent that spawns off child processes to execute the various workflow components. This agent is referred to hereafter as the root agent.

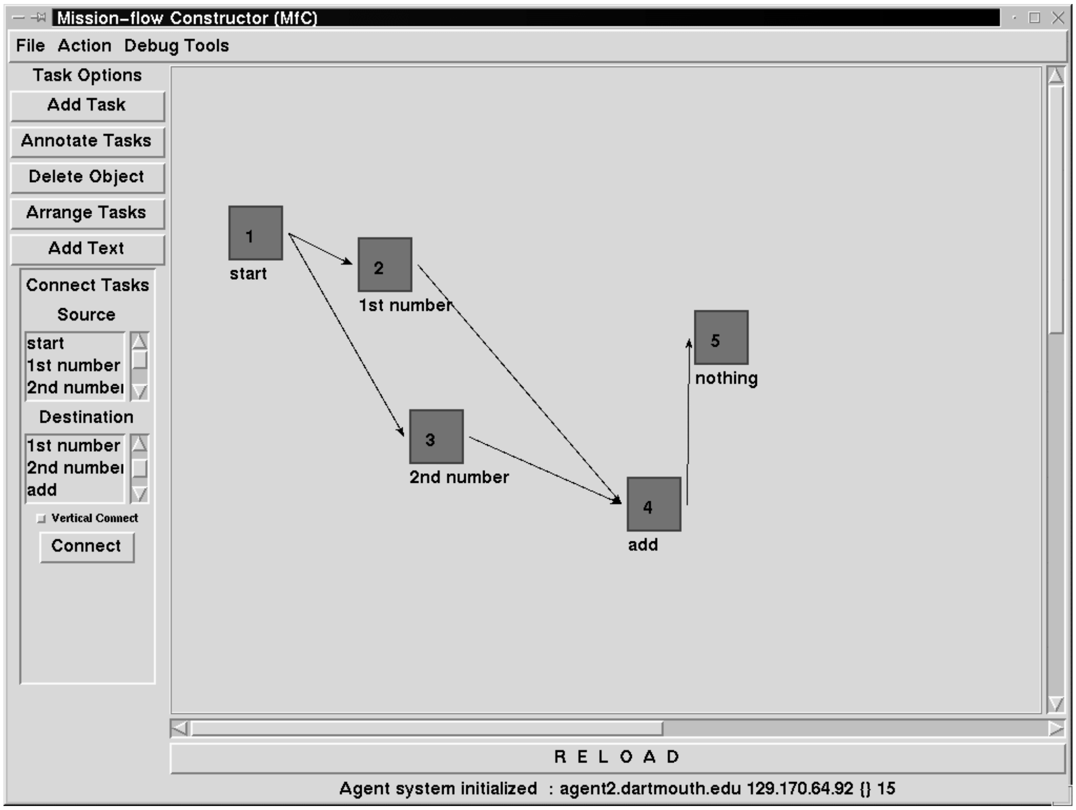
There are two distinct components that comprise MfC: the visual construction environment, and the compilation and execution engines. The visual construction environment consists primarily of a GUI that provides the user with the tools required to generate a workflow. The compilation and execution engines turn the information provided in the visual construction environment into an executable workflow and manage the actual execution of the workflow. The execution engine also implements an agent tracker that provides the user with run-time updates through the GUI. Each of these components is dealt with in detail in this section.

4.1 The Visual Construction Environment

The visual construction environment serves a two-fold purpose, the first of which is to provide the user with a means of constructing a meaningful (to the user) visual representation of the workflow. Second, to appropriate (for the compilation engine) as much information as possible from the topology drawn by the user. To this end, this part of MfC is driven (as it should be) by a graphical user interface (GUI). The graphical toolkit extensions to Tcl, i.e. Tk, make the building of a GUI a relatively simple task. The canvas found in the visual construction environment holds the set of graphical objects that provide the user with the pictorial representation of the workflow and MfC with information about the topology of the workflow. With reference to graph theoretical representation of workflows, the workflow is to be drawn as a directed a-cyclic graph. Each node in the graph drawn represents a task and each arc represents information flow. MfC allows the user to draw tasks and their temporal relationships on the canvas and also provides means of annotating the tasks with functional information. Once MfC is furnished with a topology and functional information of all tasks, the workflow has a fully specified schema and it may be compiled and then executed. When a workflow is instantiated, the GUI shows a "Tracker" window that provides real-time updates regarding the status of the various agents collaborating to execute the workflow.

4.1.1 Topological Specification: With the understanding that a workflow schema consists of both topological and functional information, MfC provides adequate means of obtaining both from the user. The visual representation is of the "box-and-arrow" form that has long been used to denote workflows. A box is drawn by clicking on the "Add Task" button found in the "Task Options" frame and then clicking on the canvas at the position the box is to be placed. The "Add Task" button binds mouse clicks within the canvas to the "*construct_box*" procedure. Once this binding is established, whenever the user clicks on the canvas, a box is drawn at that point. The *construct_box* procedure does the actual drawing of the box on the canvas. The mouse pointer's co-ordinates within the canvas are passed to *construct_box*, which draws the box at that point. This procedure also creates an entry for the task within the global variable (*tasks*) that holds information about all the tasks within a workflow.

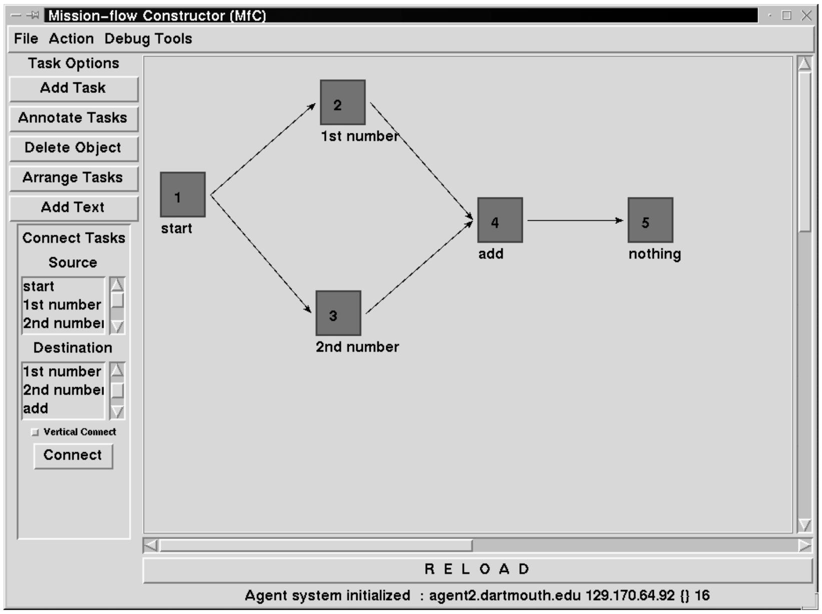
Once task boxes are drawn, their temporal relationships (and data dependencies) are to be depicted by drawing arrows between them. This is done using the tools found in the "Connect Tasks" frame. This frame consists of two list-boxes and a button labeled "Connect". Both list-boxes contain an exhaustive list of tasks in the workflow. The user selects the source tasks from one list-box and the destination tasks from the other. Once this is done, clicking on the "Connect" button draws the appropriate arrows. The "Connect" button triggers the procedure "*Connect*", which draws the arrows and adds entries to the variable *tasks* as well as the variable that holds task interconnection data (*connects*). (See Figure 6)



Agent system initialized : agent2.dartmouth.edu 129.170.64.92 {} 15

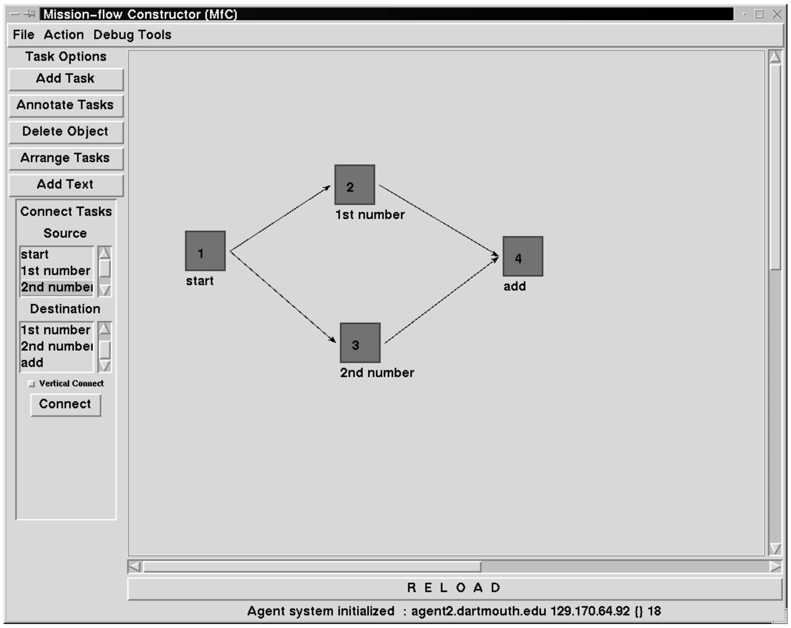
Figure 12 - Drawing a workflow in MfC

During the course of drawing a workflow, it may be required to move a task box around the canvas or delete a task box (or arrow) from the canvas. These functions are available from the “Task Options” frame as the “Arrange Tasks” button and “Delete” button respectively. The “Arrange Tasks” button binds the mouse click and drag to the “*Mark*” and “*Move*” procedures, which identify the canvas object closest to the mouse pointer and allow it to be dragged around within the canvas so that it may be repositioned. The “Delete” buttons binds mouse clicks to the “*Delete*” procedure, which removes a canvas object from the screen, as well as all of the object’s associations in the various state variables. As an example, the workflow from Figure 6 is modified using these functions and shown in Figures 7 and 8. In Figure 7, the task boxes have been moved around the canvas (for a purely cosmetic effect) and in Figure 8, the task box that does “nothing” has been deleted. All of the above functions have been adapted , with some modification, from MACE [Sha97].



Agent system initialized : agent2.dartmouth.edu 129.170.64.92 {} 16

Figure 13 - Arranging tasks in MfC



Agent system initialized : agent2.dartmouth.edu 129.170.64.92 {} 18

Figure 14 - Deleting Objects in MfC

4.1.2 Functional Specification: In addition to the topological specification described in **Section 4.1.1**, a complete workflow schema also contains functional information. The functionality of tasks is defined by task annotation. Each box on the canvas must be described using a set of predefined fields that can completely encapsulate the functionality of the task. When the user clicks the “Annotate” button, MfC binds mouse clicks to the procedure “*GetClick*”. This procedure identifies the canvas object to be annotated and pops up an annotation window (see Figure 15) that contains initial entries for the various descriptors that encapsulate the task functionality. These descriptors can be modified by the user to customize the task functionality to his/her needs. All descriptors are used to index a global array called *tasks*. These descriptors are detailed below.

Task Block Name: task4

Task Description: task4

variables available as input:
task3: resulttask3
task2: resulttask2

Figure 15 - Task Annotation in MfC

- **Name and description**: These fields provide task information to the user rather than to the workflow engine. These fields are simply used to describe the task. Both are strings and the name field cannot contain any spaces. Both of these fields are given initial values like *task1* when a task is first annotated. This is a required field.

- **Time**: This is the time limit for which the agent server on the local machine will wait for a response from the destination server before raising an exception. This is not a required field, and D'Agents provides a default value of 15 seconds if this variable is not set by the user.

- **Agent Type**: The type of the task is the primitive construct that is to be used. The annotation window varies with the type selected. The primitive constructs that are provided are *scatter*, *gather*, *sentinel*, and *decision point*. If none of these primitives are to be used, the *generic process* type can be selected. The selection of the different primitives changes some of the fields in the annotation window. The implementation of these constructs is detailed later in this chapter. This is a required field.

- **Agent Function**: A task can be purely computational or user interactive. In the case of purely computational functions, no user interface is required and the MfC will not generate a GUI for the workflow participant. This is not a required field and MfC defaults to user interactive tasks. When the user-interactive option is selected, MfC auto-generates a workflow map for the workflow participant that indicates his/her task in the workflow topology.

- **Machine Name**: This field asks for the location of the workflow participant. In the context of workflow implementation in MfC, each participant is assumed to be on a different machine in the network. The machine name field tells MfC where agent representing the task should be sent. The machine name can be either a symbolic (for example, *actcomm.dartmouth.edu*) or numeric (for example, *129.170.64.91*) IP address. This is a required field and appears when a primitive construct (agent type) is selected.

- **Username**: The user names are symbolic names assigned (by the user) to the various workflow participants who execute the different tasks. Multiple tasks can be assigned the same user and in the compilation section, we discuss how tasks associated with the same user may be traced. This is a required field.

- **Result**: In this field, the user is to enter the name of the variable that holds the result of the task's computation. MfC monitors this variable and at the end of the task's execution, sends the result data to succeeding tasks. This is not a required field. In the event that a result variable is not specified , when the task completes its function, MfC simply sends a "clear-to-start" message to succeeding tasks.

- **Code**: This field is a text box and provides the user with the means to develop a complete functionality for the task. The user is to enter Tcl/Tk code in this text box. This code is then evaluated at the location of the workflow participant. D'Agents scripts may also be entered here and they will execute correctly, however, the purpose is to allow a user who has no knowledge of mobile agent technology to define and execute a distributed workflow using mobile agents.

Using the drawing and annotation tools provided, a workflow can be completely specified and be made ready for compilation and execution. However, before discussing these functions, we provide a run-down on the way the above descriptors are stored and manipulated. Also discussed are the annotations required for the various primitive constructs.

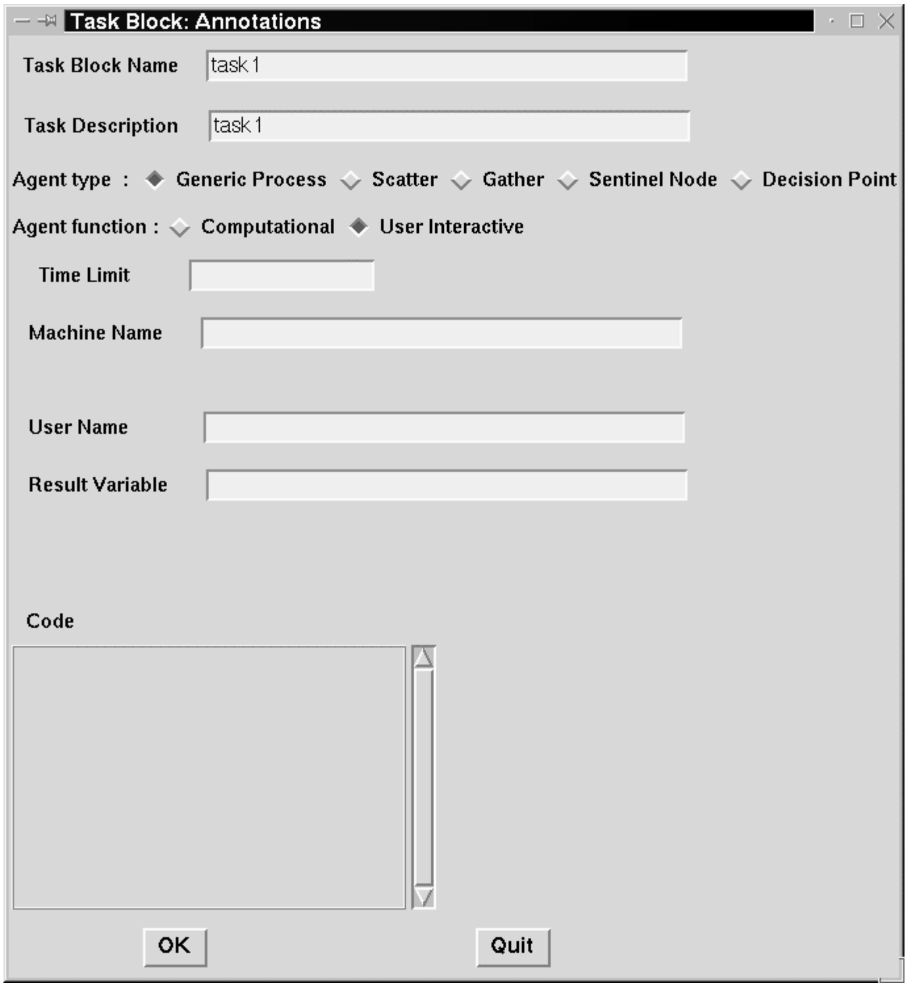
4.1.3 State Variables and Schema Capture : *Tasks* is the global variable that holds the entire workflow schema, both topological and functional. Since all variables in Tcl/Tk are treated as strings, *tasks* is implemented as an array indexed by strings. *Tasks* is an associative array, by which we mean that the indices of the array are relevant to the data stored in it. Each element of the array is indexed as *tasks(name,field)*, where *name* is the name assigned to the task and *field* is one of the descriptors listed in the previous section. Obviously, the array is associated to its contents through the name of the task and hence names must be held unique. The *tasks* array can also be thought of as a user defined data structure or "*struct*" in C. In that case, all tasks would be of the same data type (let us say *tasks*). The *tasks* data structure would then have each of the above descriptors as parameters of the variable assigned to it. Each task box would have to be defined as a separate variable of type *tasks*. The major difference is that in MfC, the fields associated with each task can be changed at will unlike a data structure in C. Since Tcl arrays do not have to be of pre-defined sizes, the *tasks* array can be written to, extended or modified during the course of execution of the MfC script. This is especially useful considering that additions to the array indices will be made whenever a task is annotated for the first time. The *tasks* array holds not only the workflow schema, but also the run-time status of the workflow. During execution, each task is also assigned a "*status*" field that indicates whether the task is active, dormant, done or dead.

As soon as a task box is created on the canvas, an entry for it in the *tasks* array is also created. This entry consists of an auto-generated name and description (both fields are given the same entry). The auto-generated entry is simply the word *task* followed by the numeric sequence in which it was created on the canvas. For example, the third task box on the canvas would be assigned the name and description of *task3*. When arcs are added to connect the various task boxes, additional entries are added to the array. The two fields added when connections are made are *input* and *output*. When the head of an arc connects to task, the source of that arc is added to the tasks *input* field. Similarly, when the tail of an arc connects to the task, the destination of that arc is added to the *output* field. The contents of these two array elements are lists. Before run-time, these are the only auto-generated fields in the *tasks* array.

During annotation, of course, the entry boxes provided in the annotation dialog provide the content of the various array elements corresponding to the task being annotated. All entries made in the annotation dialog are saved in an array called "*entries*" which is an exact mirror of the *tasks* array. The *entries* array is used so that the user can discard changes to the annotation if needed. When changes are accepted, the contents of the *entries* array are mirrored into the *tasks* array. Entries to the *tasks* array are also added during workflow execution. These include unique agent identifiers that are obtained when a task is instantiated as a remote process. The agent ids are assigned to the field *agent_id*. Also, the status of an agent that has been deployed is also held in the tasks array. It should be noted here that the entire workflow schema can be derived from the *tasks* array. Most of the procedures in MfC make use of and modify the *tasks* array. To aid debugging MfC and workflows developed in it, there exists a "Variable Dump to Screen" option in the main menu that lists the indices of the *tasks* array as well as other important variable constructs in MfC.

Another, though less critical, associative array that is used in MfC is the *connects* array. The *connects* array is used to store GUI information regarding the connecting arcs on the canvas. This array simply holds the screen id (*ID*) and canvas tag (*TAG*) of the arc and has an *input* field and an *output* field. The *input* and *output* fields are used to associate relevant task boxes with the connecting arc. Most of the other variables used in MfC are derived from the *tasks* array.

4.1.3 Annotation of Primitive Constructs: Each of the primitive constructs provided in MfC must be annotated differently because of the functional and topological constraints that they impose. It will be seen here that no constraints are imposed on the kind of functionality that tasks can offer, regardless of what primitive construct they represent. In fact, all task annotation windows have a text entry box labeled "Code" where the user may enter any Tcl/Tk code he/she chooses. Following are screen samples of the various task annotation windows and some description. Most of the figures are self explanatory, though occasional references to **Section 3.3** may be required.



Task Block Name: task1

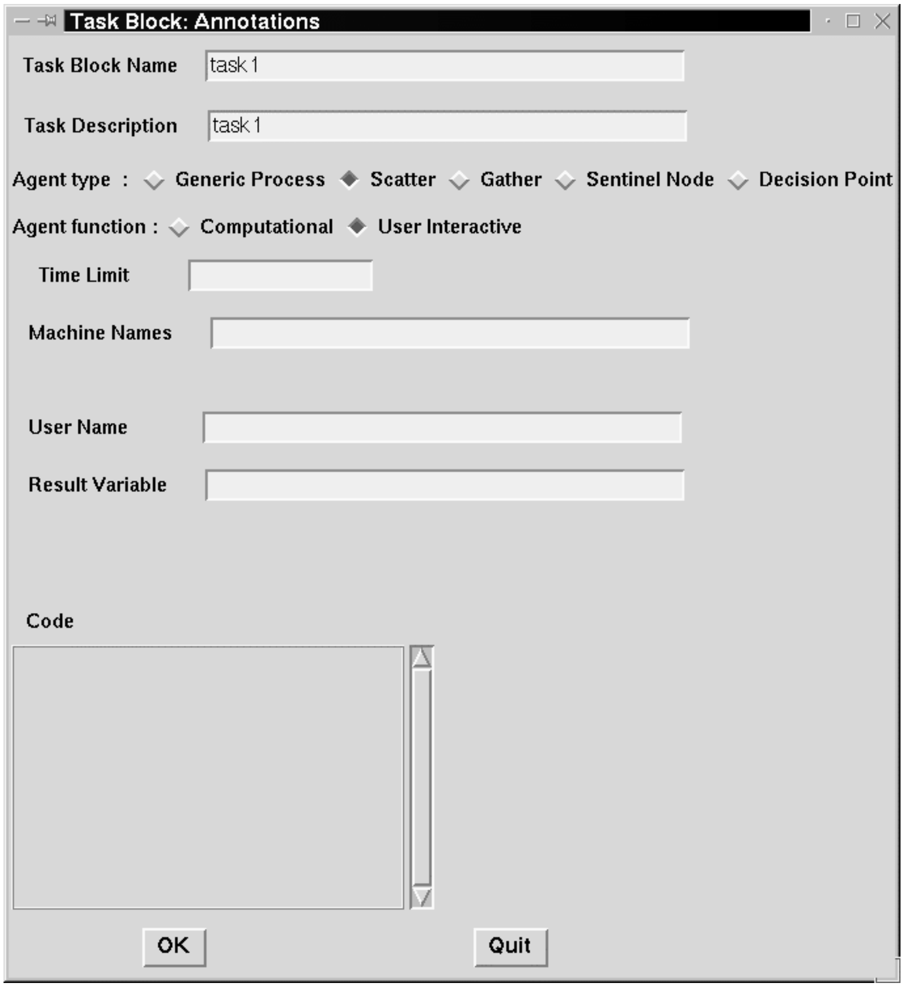
Task Description: task1

Agent type : ● Generic Process ○ Scatter ○ Gather ○ Sentinel Node ○ Decision Point

Agent function : ○ Computational ● User Interactive

Figure 16 - Generic Process Annotation Window

The generic process/task has no constraints whatsoever imposed upon its functionality nor does it have a topological import outside of the box and arrow drawing on the canvas. With this mind, the annotation is sparse with the only topological requirement being a machine name indicating the location of the workflow participant. The text entry box for the functional code requires (for successful execution) that it be Tcl/Tk code. However, a D'Agents script will also be accepted. The radio buttons that select whether the task is "Computational" or "User Interactive" are present on all annotation windows and dictate whether the agent auto-generates a GUI for the workflow participants.



Task Block Name: task 1

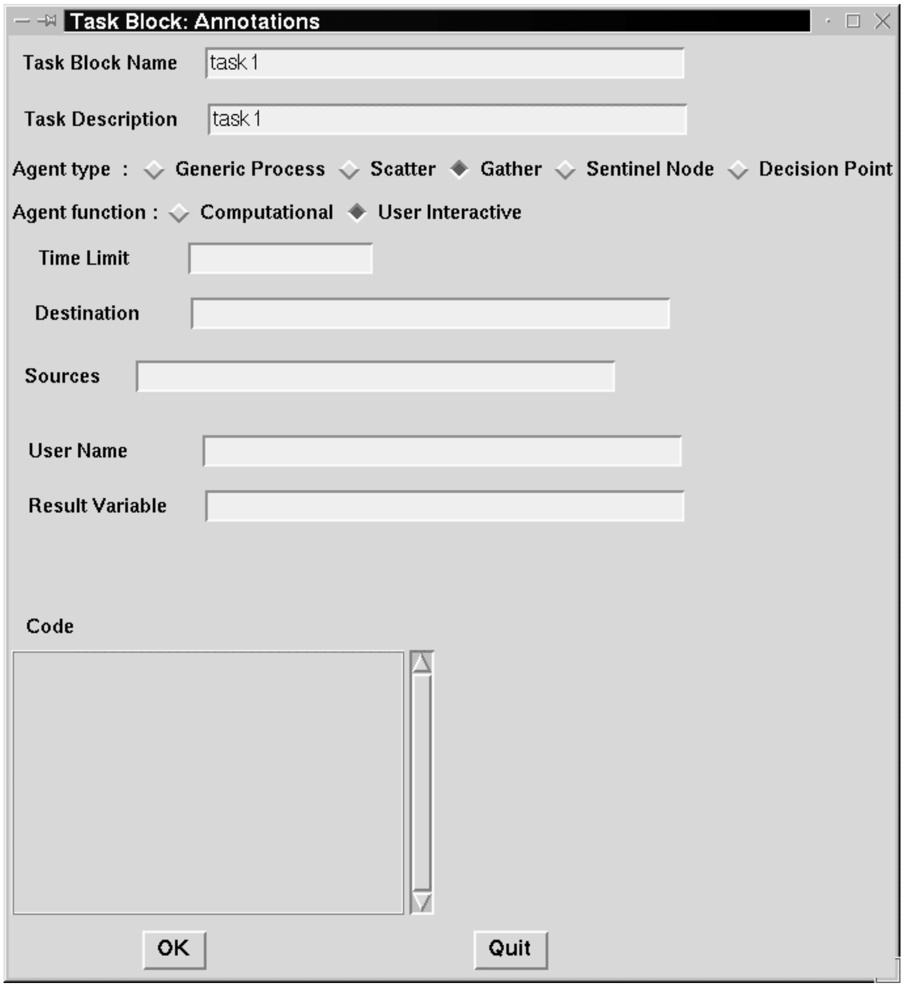
Task Description: task 1

Agent type : ○ Generic Process ● Scatter ○ Gather ○ Sentinel Node ○ Decision Point

Agent function : ○ Computational ● User Interactive

Figure 17 - *Scatter* Annotation Window

The significant difference between the *scatter* and the generic process annotation window is that the scatter primitive requires that the user provide MfC with the list of machines to which tasks must be scattered. The “Machine Names” entry box takes a list of machine addresses (either symbolic or numeric) as its argument. The code provided will then be executed at all of the remote locations specified by the user.



Task Block Name: task1

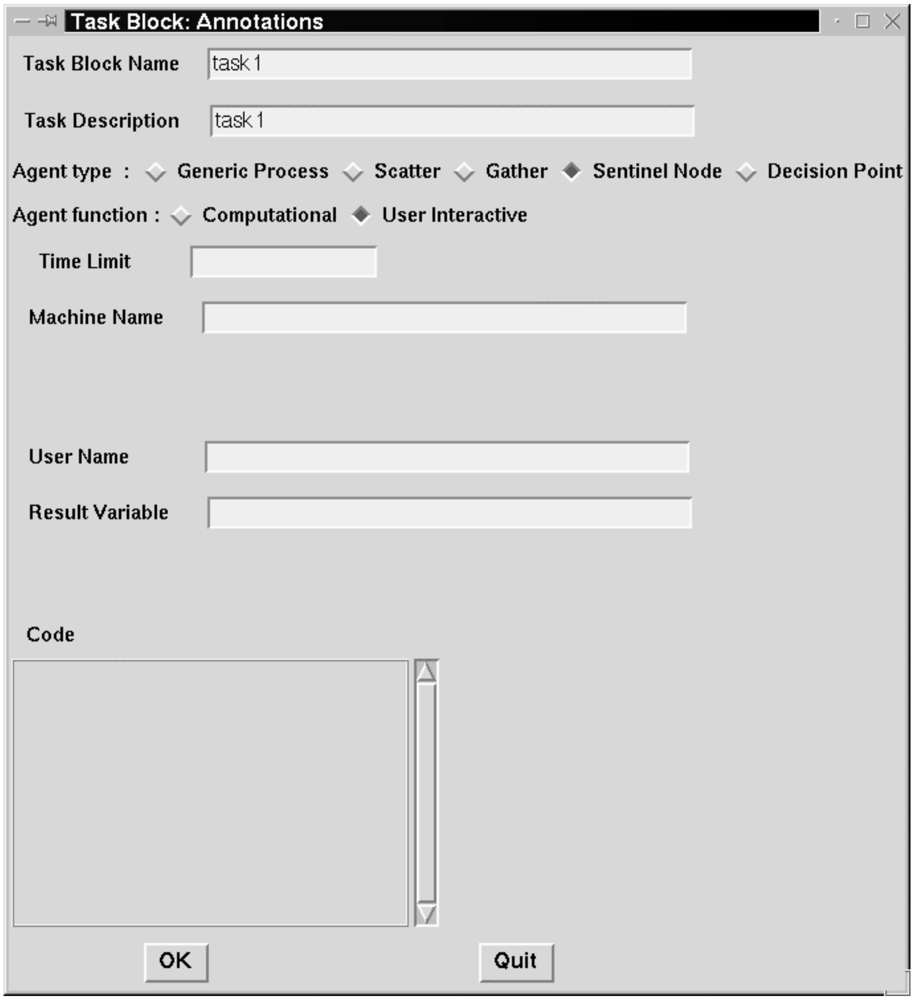
Task Description: task1

Agent type : ○ Generic Process ○ Scatter ● Gather ○ Sentinel Node ○ Decision Point

Agent function : ○ Computational ● User Interactive

Figure 18 - *Gather* Annotation Window

The *gather* annotation window requires that the user specify both sources and the destination of the task box. In this case, "*Sources*" should be the list of inputs to the *gather* operation or a subset thereof. MfC will then collate the results from the tasks listed into one variable that may be used by the code specified. The "*Destination*" field indicates the location at which the *gather* operation is to execute.



Task Block Name task1

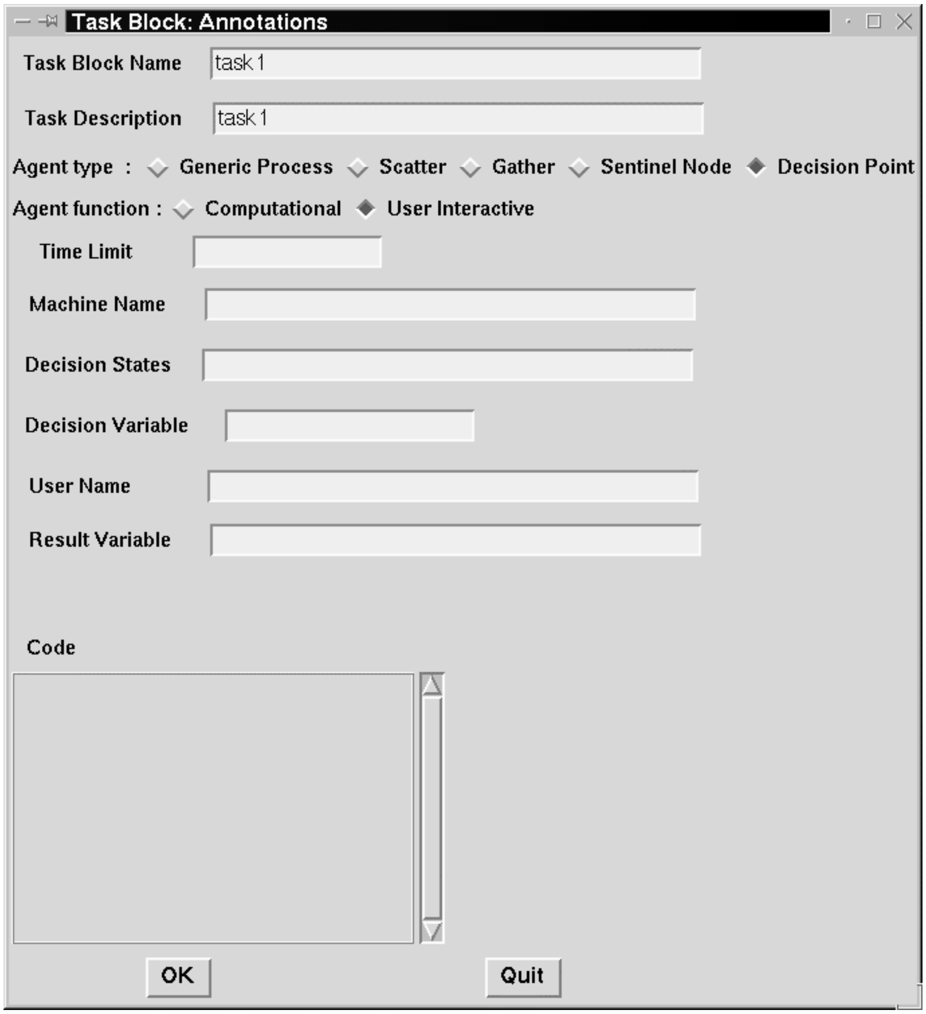
Task Description task1

Agent type : ○ Generic Process ○ Scatter ○ Gather ● Sentinel Node ○ Decision Point

Agent function : ○ Computational ● User Interactive

Figure 19 - *Sentinel* Annotation Window

The *sentinel* node is to be annotated exactly like a generic process. MfC ensures that the agent executing the *sentinel* operation does not die until the workflow is complete. Also, the repeated execution of the task is not to be coded by the user, but instead left for the execution engine to handle.



Task Block Name: task1

Task Description: task1

Agent type : ○ Generic Process ○ Scatter ○ Gather ○ Sentinel Node ● Decision Point

Agent function : ○ Computational ● User Interactive

Figure 20 - *Decision Point* Annotation Window

The *decision point* has more fields to be annotated than the other primitives available in order to maintain the flexibility that it has to offer. The “Machine Name” field specifies the location where the *decision point* object is to execute. The “Decision States” field takes a list (possibly comprised of lists) of the succeeding tasks or a subset of them. For example, if there were three tasks (*task2*, *task3* and *task4*) that succeeded the *decision point*, one possible list for the decision states field would be

{task1 task3 {task2 task1} task2}

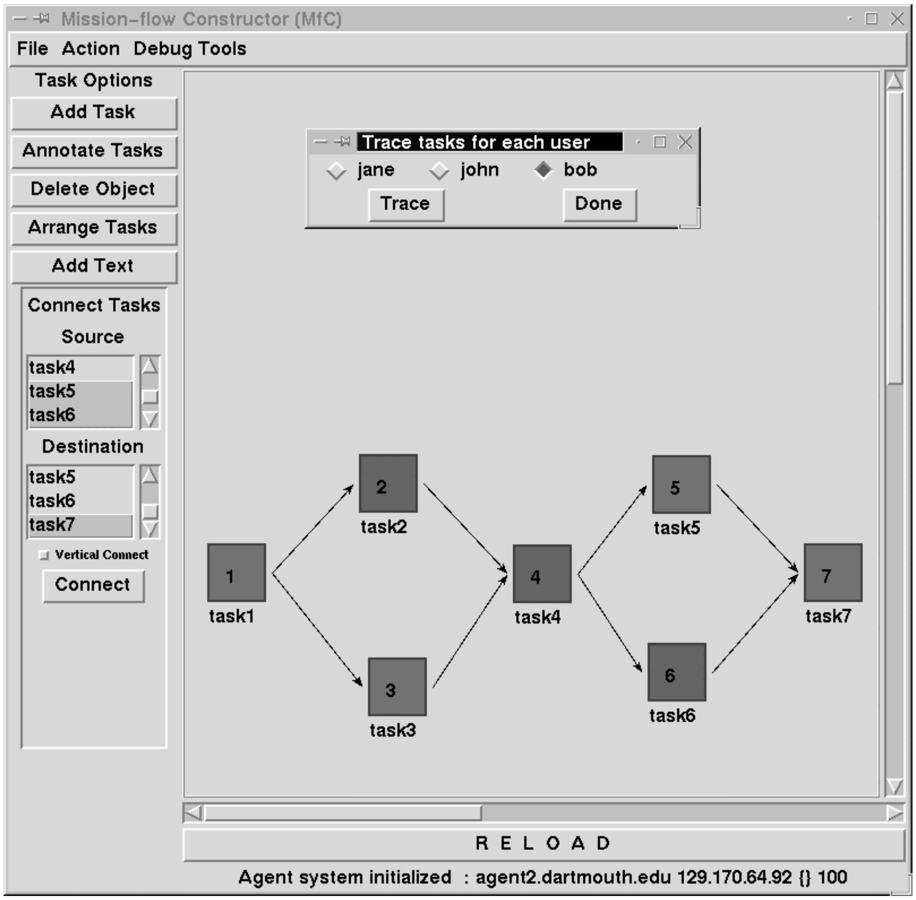
The above list shows four decision states, one of which is a list of machines. The choice of which decision state is chosen is dependent on the next field, “Decision Variable”. The decision variable must hold an integer that is not greater than the number of decision states available. The list element whose list index (list indices start from 0) is equal to the value of the decision variable will be the chosen decision state. In the above example, if the decision variable were set to a value of 2, the tasks that would be initiated would be *task2* and *task1*. The workflow engine would then kill the agent that was to execute *task3*. It should be noted here that this arrangement provides the flexibility to map multiple decision states to a smaller number of tasks. It also ensures that the same decision state can initiate multiple tasks.

4.2 Compilation and Execution

4.2.1 Compilation: Once a workflow has been specified in the visual construction environment using both the drawing and annotation tools, it has a complete schema. To execute this workflow, it is necessary to compile the schema to a set of D'Agents scripts that can be initiated at remote locations. Compilation in MfC consists of checking the workflow specification for errors, generating an error log if necessary, and identifying appropriate D'Agents wrapper for the various user specified tasks.

During the compilation process, MfC checks to see that all required fields in the annotation dialog have been given entries. When a specification error is detected, an error level is set and the error checking process continues. At the end of the error checking process, a dialog box containing the errors found is posted. If no errors are detected, MfC generates an array called *temporal_map*. This array holds the following lists: tasks that have no predecessor, tasks that have no successors, and tasks that do not fall under either of the previous categories. This array is useful for workflow initiation and for identification of workflow completion.

The most important function of the compilation engine is the enabling of user-defined tasks with D'Agents wrapper scripts. The selection of the wrapper is first decided by how the annotation dictates the task function: user interactive or computational. User interactive tasks are first assigned the *all_agents_wrapper* that automatically generates a GUI for the workflow participant at the remote location. This auto-generated GUI consists only of the recreation of the MfC canvas that holds the workflow topology. Additional wrappers are assigned based on the primitive construct that the task has been described as. Each of the primitive constructs provided in MfC require a different form of implementation, and hence the user-defined functionality must be encased in different wrappers. These wrappers are discussed in detail later in this section. When compilation is completed successfully, the compilation sets the "*compiled*" variable to indicate that the workflow is ready to be executed.



Agent system initialized : agent2.dartmouth.edu 129.170.64.92 {} 100

Figure 21 - User-centric task tracing

The compilation procedure also generates a "task tracer" when compilation has completed successfully. (See Figure 21) The task tracer is a simple dialog that highlights all the tasks associated with a particular user. Currently, the username field (though it is a required field) does not influence the execution of the workflow. In future versions, we expect to have an implicit association between a user (workflow participant) and a machine (location). This will allow the WfMS to provide location transparency. Tasks can then be forced to work in the environment of the specified user at the location. For example if user Bob was to be a workflow participant at the location *actcomm.dartmouth.edu*, the task assigned to Bob could be set to wait until Bob logs on to the machine and then execute with the permissions assigned to him. With this in mind, the task tracer has been implemented to showcase the idea that future versions will be more user-centric than location-centric. Figure 21 shows a sequence of tasks that have been assigned to the user Bob. Such sequences (successive tasks assigned to the same user) will be termed *threads*. Keeping track of such threads will be useful when developing applications such as Adaptive Active Templates (AAT) [DD99]. AATs are further discussed in the Future Work section.

4.2.2 Execution: A workflow may be executed only after it has been compiled. Once compiled, the user may initiate the workflow by choosing the "Execute Workflow" option from the "Action" menu in the main window. The "Execute Workflow" menu item is bound to the procedure "*Launch*." When called, *Launch* sends one agent for every task that has been defined to the appropriate locations. These agents are dispatched using the *agent_submit* command. All agents carry a start-up script, associated variables, and event handlers for the various messages that may be encountered. Once an agent is registered with the local agent server on the destination machine, it enters an event loop to wait for a clear-to-start message from MfC. Once all agents have registered with their local controlling servers, the root agent "broadcasts" the unique identifier of each agent to all other agents. This is done by sending the list of agent ids to all the agents that have registered. Tasks that have no predecessors are then instantiated. When a task completes execution, it sends a "done" message to the root agent. The root agent then sends clear-to-start messages to succeeding tasks with relevant result data. It should be noted here that each primitive construct has a different execution model, each of which is detailed below.

- *Scatter*: Keeping in mind that the *scatter* primitive is actually many concurrent tasks, we see that for each task that is defined as a *scatter* object, the root agent must launch as many agents as there are concurrent tasks. When a *scatter* object is called, MfC sends an agent of user-defined functionality to each of the locations provided. MfC does not keep track of the agent identifier of agents that have been created as a result of a *scatter* operation. However, MfC does keep track of the number of concurrent tasks that have completed and from the list of machine names can determine which tasks await completion. When all the child processes of a *scatter* operation have completed, the root agent sends the clear-to-start message to the successors. Tasks defined as *scatter* operations are assigned the *SCTR_wrapper* as a wrapper for the user-defined functionality. This wrapper ensures that when each child process sends a message to other agents, it is clearly identified as such.

- *Gather*: Tasks assigned the *gather* primitive use the *GTHR_wrapper* procedure to ensure such functionality. The *gather* primitive collates the result data from the tasks specified in the “Sources” list into an array indexed by those task names. The agent executes the Tcl code provided to it only when the clear-to-start message is received. It should be noted here that *gather* operation itself does not any temporal constraints enforced upon it, only the execution of the functional code. This is because the data can be gathered or collated only when it is made available by the preceding tasks.

- *Sentinel:* The sentinel agents enter an event loop as soon as they register with their local agent server. The *SNTL_wrapper* ensures that the agent remains in the event loop until an information request is received, the workflow completes, or a “terminate agent” message is received. Each time an information request is received, the wrapper evaluates the Tcl code, replies to the request with the result of the computation, and once again enters an event loop.

- *Decision Point:* When a task defined as a *decision point* is initiated, the wrapper (*DP_wrapper*) first executes the Tcl code and when the task has otherwise completed, extracts the list of tasks that correspond to the list index provided by the decision variable. These tasks are then sent a clear-to-start message while the decision states that were not chosen are sent a “terminate agent” message.

- *Generic Process:* The generic process/task option also has a wrapper on its own (*GP_wrapper*). This wrapper simply makes the agent wait until a clear-to-start message is received. Once received, the wrapper evaluates the Tcl code and returns the result variable to the root agent.

4.2.3 Communication : MfC agents are designed to communicate using short messages. Since the *agent_send* mechanism in the D’Agents system is used when raising exceptions, we use the *agent_event* and *agent_getevent* commands for inter-agent communication. However, event handlers have been established for both message-passing mechanisms. Messages are of two types, those sent through the tracker and those passed directly between agents. In the case of the former, more information from the remote tasks is required, as the tracker is the monitoring utility for the user who executes the workflow. (See Figure 22) A set of messages that provide details regarding the state of execution of each task has been implemented. These not include not only standard “starting” and “terminating” messages, but also whether the agent is waiting or has failed. In addition, each primitive construct has a unique list of informative messages that enable effective monitoring.

Under normal circumstances, all communication passes through the tracker so that the user who executes the workflow is able to monitor (during run-time) the status of the workflow. However, this also makes the location of the tracker a focal point of failure. In the event that the tracker goes off-line, communication between agents is abruptly cut off. In order to function effectively even without the tracker and root agent, the agents must be able to seamlessly change communication channels. That is, agents must start communicating directly amongst themselves. When the agents were first deployed, the root agent (or tracker) provided each agent with the global ids of all the other agents. In the event of tracker failure, agents can route messages directly to each other using these identifiers. The deployed agents recognize tracker failure when they cannot establish communication with the tracker. The first agent that recognizes tracker failure sends a message to that effect to all the agents that succeed it. Agents that receive the tracker failure message pass the same message to their successors. Thus, all agents that need to communicate with the tracker are kept advised of the tracker's status. Once an agent receives a message that the tracker has failed, it sets up new (though predefined) message handlers to handle incoming messages and routes result data directly to the succeeding agents rather than to the tracker.

```
Agent registered! ack_msg_3

Agent registered! print

Agent registered! combine_msgs

Agent registered! ack_msg_1

Agent registered! send_msg

ack_msg_3 is waiting for previous task(s) to complete

print is waiting for previous task(s) to complete

combine_msgs is waiting for previous task(s) to complete

ack_msg_1 is waiting for previous task(s) to complete

send_msg has begun

send_msg has terminated

ack_msg_3 has begun

ack_msg_1 has begun

ack_msg_1 has terminated

ack_msg_3 has terminated

combine_msgs has begun

combine_msgs has terminated

print has begun

print has terminated
```

Figure 22 - Agent Tracker

Chapter 5

Conclusion

In this thesis, we have presented the concept of a mission – a distributed dynamic workflow – as well as the need for a completely flexible workflow management system. Requirements for the design and implementation for such systems were discussed in some detail. We have implemented Mission-flow Constructor (MfC), a workflow management system that provides a user with the ability to define, execute and monitor a distributed dynamic workflow. MfC abandons the traditional single-agent approach to implementing workflows and instead, uses a larger number of small agents. In the implementation of MfC, we provide primitive constructs in workflow specification that are neither strictly topological nor functional. These constructs drastically reduce the amount of time taken to code repetitive functions. We have also developed some basic fault tolerance mechanisms towards network failure. MfC demonstrates significant improvement over its predecessor MACE [Sha97] in terms of ease and depth of workflow specification, efficient use of network bandwidth, inter-agent communication and fault tolerance.

Chapter 6

Future Work

As with most theses, there is a large body of work related to the current work that could not be completed for reasons such as the scope of the thesis and time constraints. Some of the work that the author hoped to do as well as a few suggestions regarding the future direction of this project are enumerated below. Of course, this cannot be a comprehensive list of all possible and necessary improvements, nor is it intended to be.

Adaptive Active Templates (AAT): When multiple tasks (usually in a linear sequence) are linked to the same workflow participant, they form what can be thought of as a "thread". Many transactional operations involve a set of small tasks (filling a form, evaluating credit history, etc.) that are usually assigned to the same workflow participant. Each such sequence would be a "thread" for that particular participant. In many cases, the functionality of some of these tasks depends on the result of preceding tasks within the thread, and occasionally on tasks that lie outside the thread. With the form-based nature of transactional and strategic operations, each thread could be modeled as an active template that adapts its interactive components based on the current state of its thread as well as that of related threads [DD99]. When one moves from the paradigm of workflow management systems to adaptive active templates, the implementation strategy changes only slightly. One important consideration here is that all tasks that are assigned to the same workflow participant could be collated into one agent whose functionality depends on the results of interaction with the user.

Agent Construction Environment: With the ideas of reusable code and ease of specification in mind, it would be useful to provide an environment wherein task specification is the focus. Here, a user can build a task that may be many times and add it to the "library" of tasks available within MfC. Annotation of tasks within MfC then becomes simpler because of the availability of a number of predefined tasks. The

Agent Construction Environment would include debugger so that the task functionality can be verified before it is committed to the repository. A debugger for D’Agents called AGDB has already been developed at Dartmouth College [HK97]. The idea of a library of agents lends itself to the obvious extension that there should also exist a library of commonly used topologies, akin to the primitive constructs provided in MfC albeit more complex.

Critical and Non-Critical Inputs: The current implementation of MfC uses a “finish-to-start” model of task instantiation. This means that for a given task to commence execution, all tasks immediately preceding it must have completed execution. This leads one to conclude that the successful completion of all previous tasks is critical to the execution of the task under consideration. This is not always true in many real world applications where inputs to a given task are to be treated on an “if-available” of “if-possible” basis. The successful execution of such tasks is not critical to the success of the mission. It is desirable to be able to annotate certain tasks, and hence the output arcs of these tasks as either “critical inputs” or “non-critical inputs” to their successors. Once a mission is instantiated, a task would then wait only until all critical inputs had been filled before commencing execution. This however, raises the question of how to handle non-critical inputs that arrive after task execution has begun.

Cyclic graph structures: Currently, MfC does not handle cyclic graphs. It is left to the user to spot cycles within the mission topology and remove them. To reject cyclic graphs outright would not necessarily be a good idea considering that fact that many applications involve the repetition of a set of tasks until a certain condition is met. It would be advantageous to provide MfC with the capability to identify cyclic sub-graphs and prompt the user to identify the “starting node” of the cycle. At that point, it would be necessary to identify a certain set of inputs to the starting node as the critical inputs for that node. The remaining non-critical inputs are to be treated as such for the first iteration of the cycle only, after which the critical and non-critical inputs are either reversed or redefined completely.

GUI Improvements: In this work, function has been placed above form when designing the GUI. While this means that all available functionality is present in the GUI, it has come at the cost of ease of use. In spite of the fact that GUI improvement is usually cosmetic, its need becomes apparent with repeated use of the application. In addition, semantic and functional content are not currently available from the visual representation of the workflow. Means of providing such content would be extremely helpful. Regardless of functionality, all task boxes look alike. Task boxes could be depicted differently based on the functionality they offer. The first step would obviously be to visually differentiate the various primitive constructs. Some additional menu options, such as those found in commercial applications would be useful.

Chapter 7

References

[Bil99] Bilar, Daniel J., ‘*Process Automation and Agents*’. Presentation, D’Agents Research Group Meeting, Dartmouth College. February 1999

[CGN96] T. Cai, P. A. Gloor, S. Nog, ‘*DartFlow: A Workflow Management System on the Web Using Transportable Agents*’. Technical Report PCS-TR96-283, Dartmouth College. May 1996. ftp://ftp.cs.dartmouth.edu/TR/TR96-283.ps.z

[CHK94] D. Chess, C. Harrison, A. Kershenbaum, ‘*Mobile Agents: Are They a Good Idea?*’ IBM Research Report RC 19887 (88465), T. J. Watson Research Center. December 1994

[CHRW98] A. Cichocki, A. Helal, M. Rusinkiewicz, D. Woelk, ‘*Workflow and Process Automation: Concepts and Technology*’. Kluwer Academic Publishers © Kluwer Academic Publishers 1998

[DD99] Dyer, Maj. Doug. “*Active Templates: Winter PAD Review – Dynamic Spreadsheets for Planning and Execution*” Information Exploitation Office, Information Systems Office, DARPA. http://www.darpa.mil/iso/act/act_brief.ppt

[GBCK94] E. P. Glinert, M. M. Blattner, S. Chang, O. J. Kurlander. Panel: ‘*Visual Languages and Programming in the Year 2004*’. In *Proceedings of the 1994 IEEE Workshop on Visual Languages*. IEEE, September 1994

[Gra95] Gray, Robert S., ‘*Agent Tcl: Alpha Release 1.1 Documentation*’. Dept. of Computer Science, Dartmouth College. December 1995

[Gra96] Gray, Robert S., ‘*Agent Tcl: A Flexible and Secure Mobile Agent System*’. Appeared in Proceedings of Fourth Annual Usenix Tcl/Tk Workshop, pp.9-23. 1996 http://www.cs.dartmouth.edu/~agent/papers/tcl96.psp.z

[Gra97] Gray, Robert S., ‘*Agent Tcl: A Flexible and Secure Mobile Agent System*’. Ph.D. Thesis. Dept. of Computer Science, Dartmouth College. 1997 ftp://ftp.cs.dartmouth.edu/TR/TR98-327.pdf

[GCKR96] R. S. Gray, G. Cybenko, D. Kotz, D. Rus, '*Agent Tcl*'. Dept. of Computer Science, Dartmouth College. May 1996

[HK97] M. Hirshl, D. Kotz, '*AGDB: A Debugger for Agent Tcl*'. Technical Report PCS-TR97-306, Dept. of Computer Science, Dartmouth College. February 1994

[IBM] IBM, "*Flowmark*" Software product. http://www.software.ibm.com

[Joo95] Joosten, Stef, '*Process-Definition-Organization Framework for Analyzing Workflow Management Systems*'. Computer Information Systems Dept., Georgia State University. December 1995

[Kob97] Kobielus, John G., '*Workflow Strategies*'. IDG Books Worldwide, Inc. © IDG Books Worldwide, Inc. 1997

[Kou95] Koulopoulos, Thomas M., '*The Workflow Imperative: Building Real World Solutions*'. International Thomson Publishing Inc. © T. M. Koulopoulos 1995

[MLL97] M. Merz, B. Liberman, W. Lamersdorf, '*Using Mobile Agents to Support Inter-Organizational Workflow Management*'. Distributed Systems Group, Computer Science Dept., Hamburg University, Hamburg, Germany. November 1997

[MN] M. Marazakis, C. Nikalaou, '*Towards Adaptive Scheduling of Tasks in Transactional Workflows*'. Dept. of Computer Science, University of Crete and Institute of Computer Science, FORTH.

[Naj94] Najork, Marc A., '*Programming in Three Dimensions*'. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1994. http://www.research.digital.com/SRC/personal/najork/thesis/thesis.pdf

[Ous94] Ousterhout, John, '*Tcl and the Tk Toolkit*'. Addison-Presley Professional Computing Series, © Addison-Presley 1994

[Sha97] Sharma, Rohit, '*Mobile Agent Construction Environment*'. MS Thesis, Dartmouth College, 1997

[Wel95] Welch, Brent, '*Practical Programming in Tcl and Tk*'. © Prentice Hall 1995

[WF94] T. E. White, L. Fisher (editors), '*New Tools for New Times: The Workflow Paradigm*'. Future Strategies, Inc. © Future Strategies, Inc. 1994

[WfMC95] The Workflow Management Coalition, '*The Workflow Reference Model*'. Document Number TC00-1003, © The Workflow Management Coalition1995

[WfMC00] The Workflow Management Coalition, '*Workflow Standard – Interoperability Wf-XML Binding*'. Document Number WFMC-TC-1023, © The Workflow Management Coalition 2000

[Zim98] Zimmerman, Daniel M., '*A Preliminary Investigation into Distributed Dynamic Workflow*'. MS Thesis, California Institute of Technology, 1998