



Kansas State University
234 Nichols Hall
Manhattan, KS 66506-2302

Phone: (785) 532-6350
Fax: (785) 532-7353
<http://macr.cis.ksu.edu/>

Technical Report

**Distributed Efficient Multi-Robot Cooperation
Framework (DEMiR-CF) in an Object
Construction Application**

by

Mike Fraka

MACR-TR-2010-06

August 31, 2010

1 Introduction

This paper documents the Distributed Efficient Multi-Robot Cooperation Framework (DEMiR-CF) [1] in the context of an object construction application replicated from Sariel and Balch [2]. First, I discuss the DEMiR-CF framework with an overview, implementation assumptions, design overview, a discussion of how to use the framework, and conclude with an overview of the JUnit test cases for the framework and its output. Second, I describe the application, discuss the classes and interfaces I was responsible for at the execution component level and below, and finish with how to run the application and application test results.

2 DEMiR-CF Framework

2.1 Framework Overview

The DEMiR-CF framework provides the means for a cooperating group of agents to achieve common goals by efficiently allocating tasks [1]. The agents in DEMiR-CF are assumed to have certain capabilities and the tasks describe the capabilities required to execute them. The tasks may or may not require multiple agents to be executing the same task simultaneously to be achieved. The tasks may or may not be independent of each other. In our version of the object construction application discussed below, the tasks require only a single agent but have dependencies between some of the tasks.

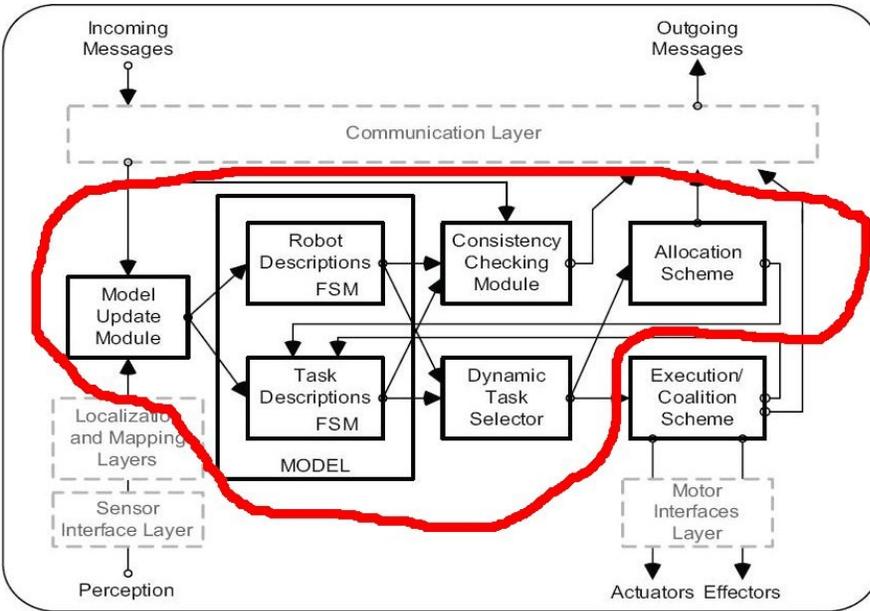


Figure 4.1: DEMiR-CF Modules

Illustration 1: DEMiR-CF System Boundary

The DEMiR-CF framework consists of the modules shown in Illustration 1 above [1]. The red boundary superimposed on the figure encloses the modules that are included in this implementation. This boundary excludes only one module from the original Sariel version, the Execution/Coalition Scheme. I excluded this module because we decided to not support coalitions of agents executing the same task in our application. The reasons for this are that GMoDS and OMACS currently do not support tasks that require more than one agent to execute, the coalition aspects of DEMiR-CF are not as well documented or straightforward as the single agent aspects, and the time available to do the project was limited.

As one can see from the figure, the framework sends and receives messages to achieve the task allocation and so assumes the existence of an external communication layer (see [1] p. 46 for a description of the messages sent by DEMiRCF; only a subset were needed for non-coalition tasks). In addition, the framework assumes sensors and localization and mapping modules exist that can provide the location of the agent to the framework. The final external module assumed by the framework is labeled the "Motor Interfaces Layer" in this diagram; in my implementation the framework assumes that an `Agent` interface is provided that can be told to execute, cancel, or estimate achievement time for a given task and that these actions will be implemented by the situated agent.

The Model Update module receives input in the form of messages and sensor/location input. The Model Update module passes the messages to the System Consistency module (which uses "Plan B Precaution Routines" [1]) to assure that there are no inconsistencies in this agent or other agent's knowledge of achieved tasks implied by the message. If the message is consistent, it is applied to the Model module to update the world knowledge of the agent of other agents (Robot descriptions) and tasks (Task descriptions). The System Consistency module issues a Warning message to other agents if an inconsistency is detected. The Model module is implemented using finite state machines (FSMs) to represent the state of an agent or task (see [1] pp. 50-52 for a description of the states and transitions of these FSMs). Certain state changes or the addition of a newly discovered task will trigger the Dynamic Task Selector Module to apply a prioritization scheme (Dynamic Priority-based Task Selection Scheme (DPTSS)) to select the optimum task for this agent. (Note: there is some variation in the algorithms that comprise DPTSS across the papers published by Sariel; most of this implementation of DPTSS comes from [3], but the "Generate Priority List" algorithm prefers tasks already awarded to this agent over free (Uncertain/Available) tasks per [2] (p. 5 Figure 4 Action Selection.)) If the optimum task is not currently being executed by the agent, the Allocation Scheme module initiates an auction for the task. All agents that receive the Auction message and are capable of the task bid on it. The auctioneer selects the lowest cost agent (considering its own bid) and awards that agent the task. If the auctioneer receives a timely Confirm message the auction is completed and the task will begin execution by the awarded agent. If the auction doesn't succeed it is canceled. Several other rules apply for consistency and optimum cost that could result in an auction or executing task being canceled ([1][2][3]).

2.2 Implementation Assumptions

I made several assumptions during the implementation of DEMiR-CF. Some assumptions were necessary due to use of GMoDS/OMACS as the source of tasks, some because of variation in the documentation of DEMiR-CF across the available papers, and some were needed because of perceived lack of clarity in the thesis.

2.2.1 Task Dependencies Will Be Observed in GMoDS not DEMiR-CF.

DEMiR-CF tasks as represented by Sariel and Balch [2] include hard and soft dependencies on other tasks. A hard dependency requires that task A finish before task B can begin. A soft dependency allows task A and task B to execute in parallel but task A must finish before task B can finish. In our application, GMoDS imposes hard dependencies via "precedes" relations and does not represent soft dependencies. To accommodate soft dependencies, we split tasks in Sariel and Balch's application into two tasks with the second of the tasks prevented from activating until a precedes relation is satisfied. GMoDS issues tasks to DEMiR-CF only when they are active in GMoDS.

2.2.2 Valid Execution Message Model Update Specified in Table 4.6 is Redundant/Inconsistent.

Sariel's thesis [1] (Table 4.6 p. 53) states that a valid EXECUTION message should result in the action "If there are other tasks with state of *self_inexec*, these states are change to *uncertain*". The first problem is that there should only be one `Task` in the `self_inexec` state. Second, this stated action appears inconsistent with other

parts of the thesis. The condition where another agent is executing the same task as this agent is handled by Table 4.4 "Precautions for contingencies and conflicts" on p. 52 of [1] in the entry "A task being executed/auctioned is announced as being executed auctioned". In this table, the lowest cost agent continues executing the task. Other rules apply for determining which auctioneer should continue, such as the auctioneer with the most agent models or the lowest unique identifier [2] (p.5). Thus, the Table 4.6 entry appears redundant.

2.2.3 An Optimum Task Already Awarded to This Agent Requires No Auction.

If the DPTSS algorithm re-selects as the optimum task for this agent, a task that has already undergone an auction and as a result was awarded to this agent, then no auction need be performed again. This could happen if the Agent is finishing up a task but bid on and was awarded another task. The "Generate Priority List" step of the DPTSS algorithm uses a priority queue that gives preference to tasks already awarded to this agent over "free" (*available/uncertain*) tasks as the primary sort criterion and cost as the secondary sort criterion.

2.2.4 CancelExecMsg Transitions Uncertain State to Available.

Sariel's thesis [1] (Table 4.4 p. 52) states that a "cancellation message received for a task being executed by the sender robot" should result in "the task and robot states are set as available and idle, respectively". However, the thesis does not address receiving a CancelExecMsg while the task is in the uncertain state. I assume that the task model should transition to available if such a message is received.

2.2.5 Resource Capacity Should Be Included In DPTSS.

Even though our application will not play resource capacity, I included it as a consideration of DPTSS. It will be played if a task has a non-zero required capacity or effectively ignored otherwise.

2.2.6 Tasks Requiring Multiple Agents to Execute Can be Omitted.

As described above, we decided to exclude coalitions from this version of DEMiR-CF. As a result of this omission, multiple robot task models and several message types were not required in addition to the Execution/Coalition Scheme module. However, the main difficulty to extend DEMiR-CF for coalitions will not be the multiple robot task models or messages but gaining an understanding of what the Execution/Coalition Scheme module does since this is not explained in much detail in the Sariel papers ([1][2][3]).

2.3 Design Overview

2.3.1 Architecture

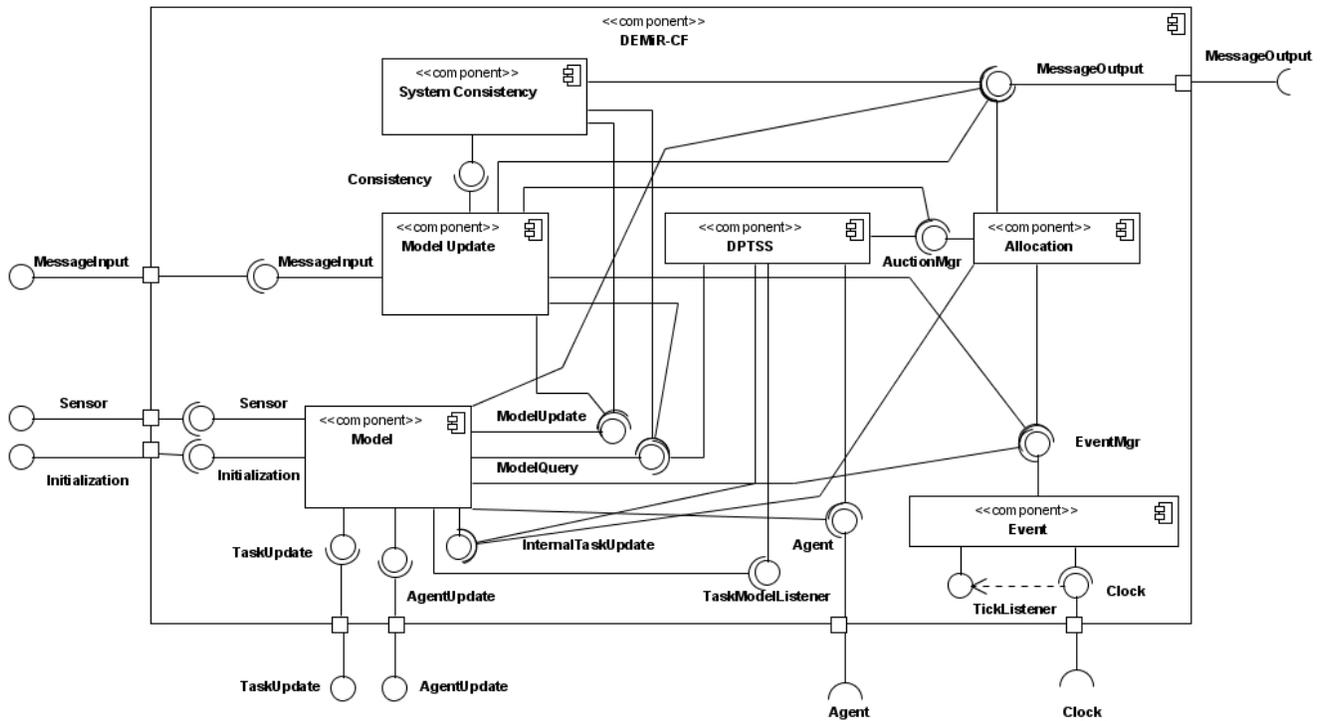


Illustration 2: DEMiR-CF Architecture

Illustration 2 above shows the DEMiR-CF framework architecture in terms of components and their interfaces. The following sub-sections describe the interfaces shown in this diagram. The next section describes in more detail the components on the diagram that implement the various interfaces and the class responsibilities within the components.

2.3.2 Interfaces

The interfaces described below are categorized as external provided, external required, and internal.

2.3.2.1 External Provided Interfaces

Illustration 3 below shows the externally provided interfaces of the DEMiR-CF object.

The Initialization interface provides methods to add tasks and agents to DEMiR-CF without triggering any task allocation. Thus, any task that should not be auctioned and instead must be executed by every agent should be given to DEMiR-CF using the `addTask` method. The `addAgent` method could be called if the client knows other agents before any `demircf.Message` is received.

The client shall call the AgentUpdate interface method `agentFailed` if the client fails. This prevents DEMiR-CF from sending any Messages. The `agentRecovered` method should be called if the client recovers.

The client must call the `TaskUpdate` interface method `newTask` for each task that should be auctioned. The client shall call the `taskAchieved` when the client has achieved an assigned task. The client calls `taskReadyForExec` when the client has received the assignment and is ready to begin executing that task. The client shall call `taskUnachievable` if the client has found that the task is infeasible. The agent calls `taskExecutionCanceled` when the agent has been given a higher priority task and must cancel execution of the current task.

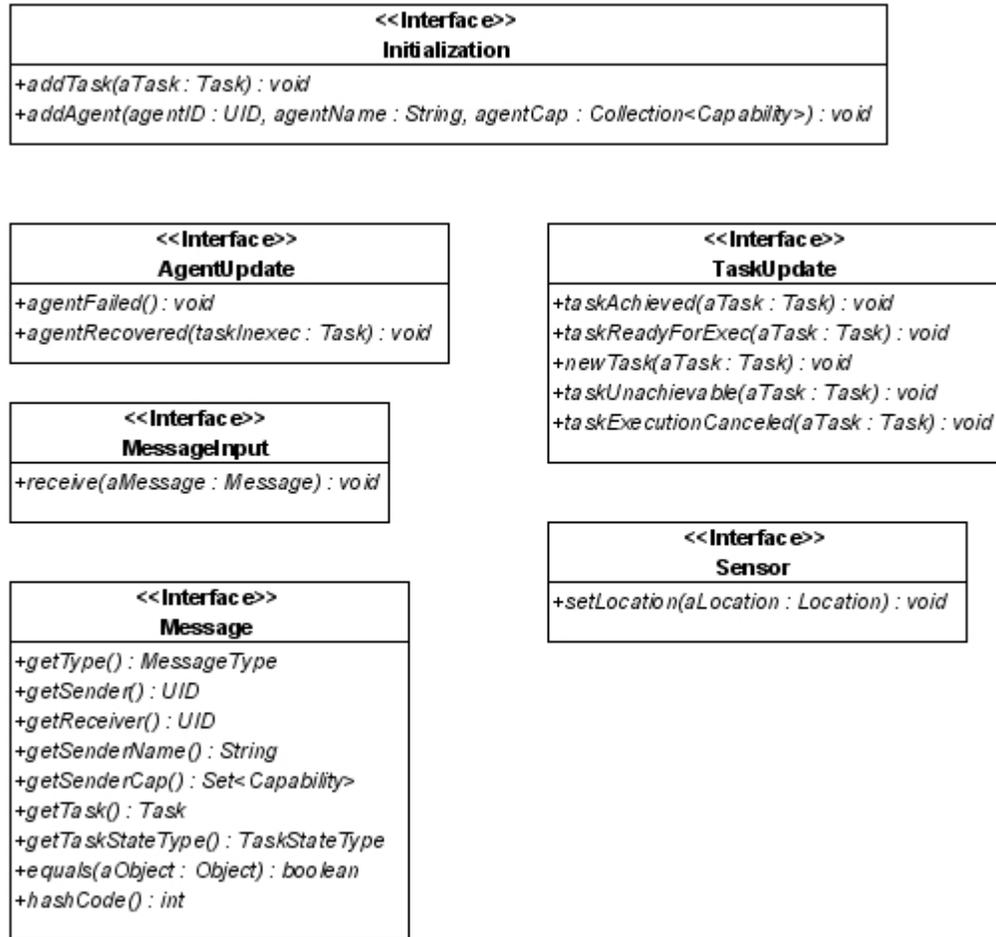


Illustration 3: DEMiR-CF External Provided Interfaces

The agent calls the `MessageInput` interface method `receive` whenever the agent receives a `demircf.Message`.

The client calls `Sensor` interface `setLocation` on each simulation turn.

2.3.2.2 External Required Interfaces

Illustration 4 below shows the interfaces required by DEMiRCF. A client must provide an implementation of these interfaces (except for `Capability`).

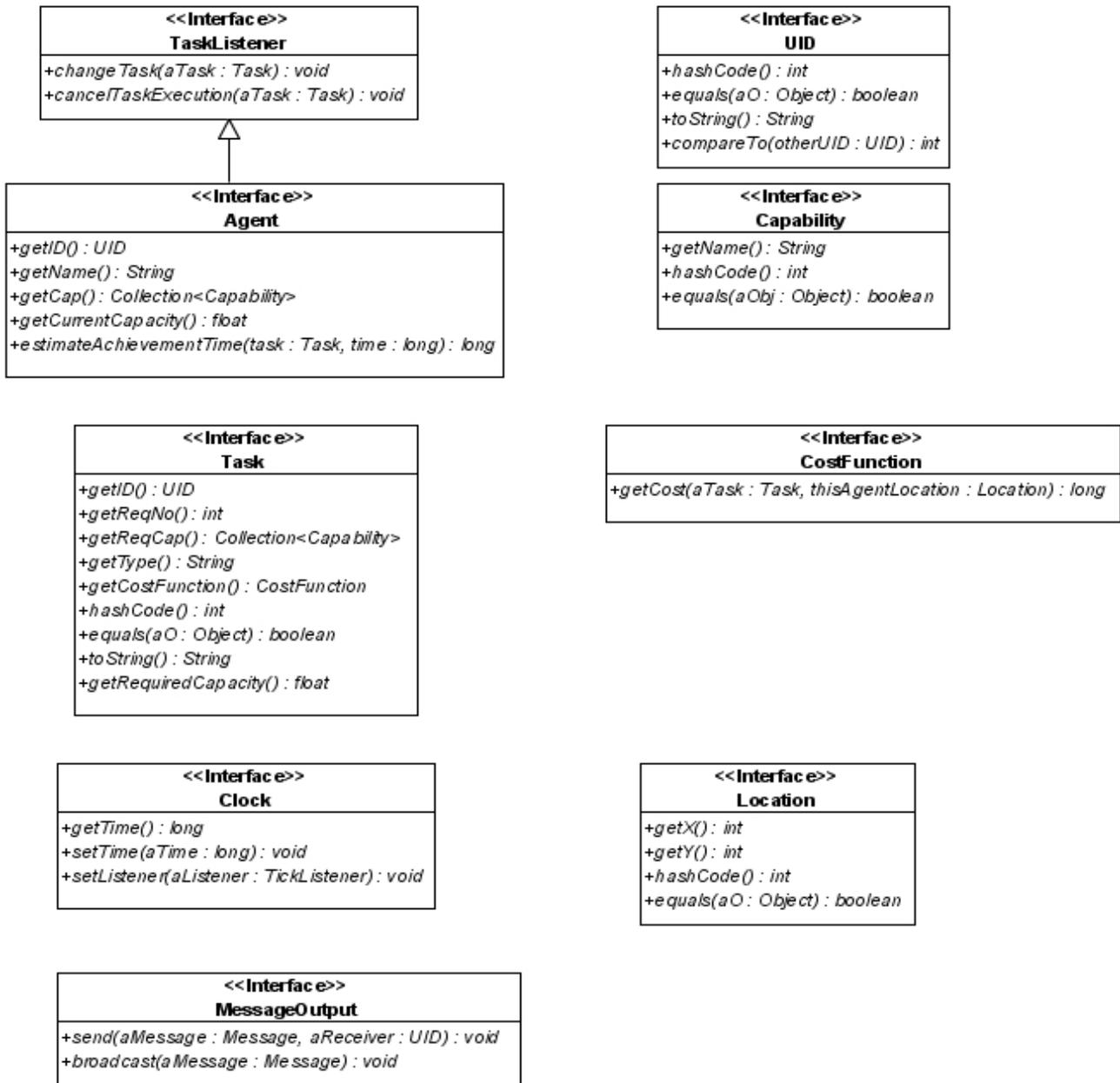


Illustration 4: DEMiR-CF External Required Interfaces

DEMiR-CF uses the `TaskListener` and `Agent` interfaces to get status from the client and to inform the client of task changes and cancellations.

The `UID` interface provides an identifier object to DEMiR-CF for `Agents` and `Tasks`.

The `Capability` interface does **not** need to be implemented by the client. Instead, the client must register its capabilities by calling the static method `DEMiR-CF.createCapability` passing in the string name of the capability and receiving a `demiRCF.Capability` object to hold in the `Agent` or `Task` implementation.

2.3.2.3 Internal Interfaces

Illustration 5 below shows the first half of the internal interfaces between DEMiR-CF modules.

The `Event` module provides the `TickListener` interface used by the `Clock` interface in order to trigger `Events` that cause regular timed behaviors in `DEMiRCF` such as timeouts. The `Event` module provides the `EventMgr` interface used by several modules to schedule and cancel `Events`.

The `Model` module provides the `InternalTaskUpdate` interface used by the `DPTSS` and `Allocation` modules to inform it of `Task` state changes.

The `DPTSS` module provides the `TaskModelListener` interface used by the `Model` module to trigger task selection and inform `DPTSS` of state changes.

The `Allocation` module provides the `AuctionMgr` interface used by the `DPTSS` module to initiate and manage `Auctions` to allocate `Tasks` in an optimum manner.

The `Consistency` module provides the `Consistency` interface used by the `ModelUpdate` module to check the validity of `demircf.Messages` received by it.

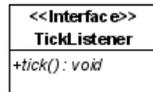
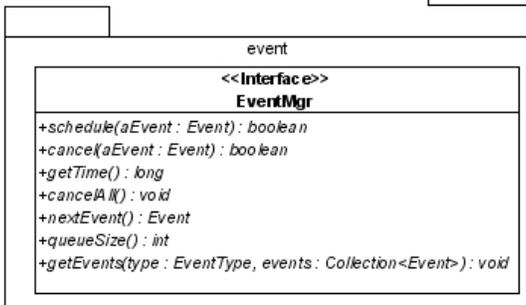
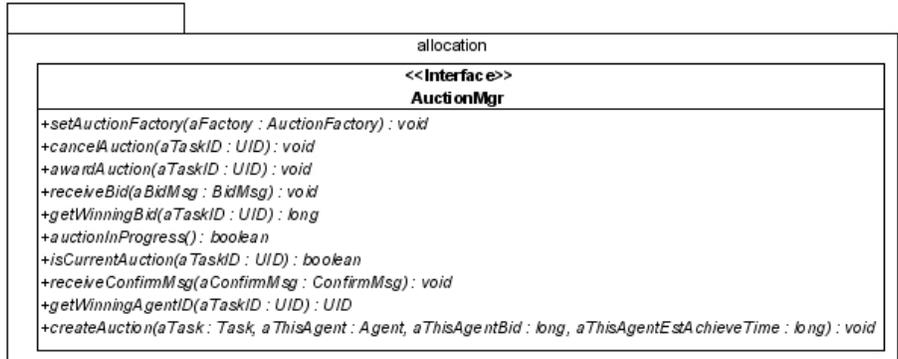
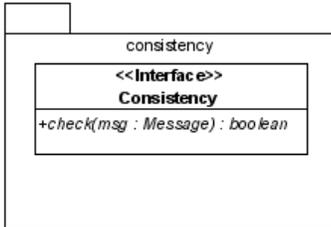
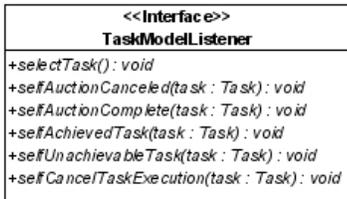


Illustration 5: DEMiR-CF Internal Interfaces Part 1

Illustration 6 below shows the remaining internal interfaces.

The Model module provides the ModelUpdate and ModelQuery used by several modules to obtain and update the world knowledge of DEMiRCF.

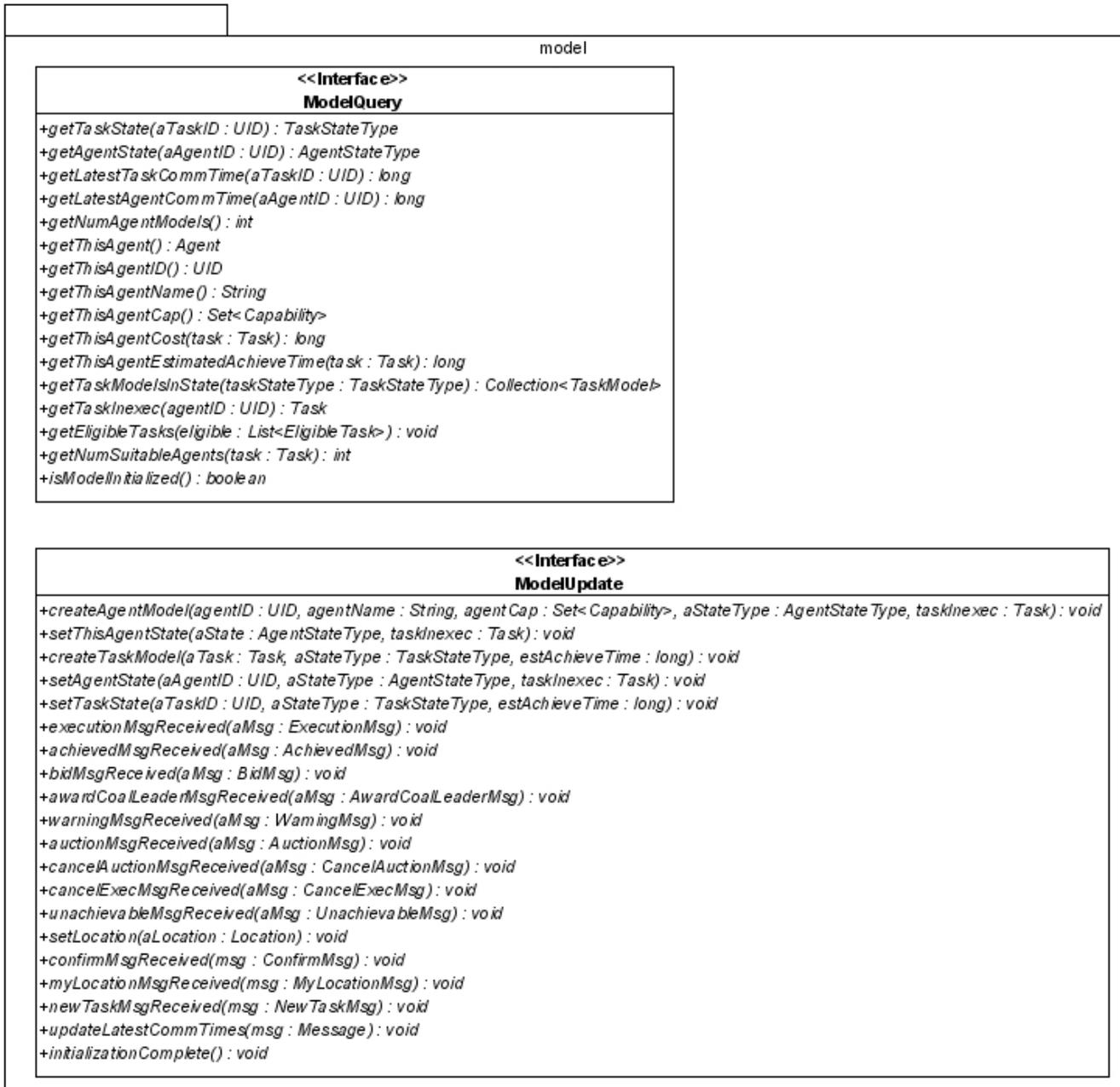


Illustration 6: DEMiR-CF Internal Interfaces Part 2

2.3.3 Components

This section describes the internal components of DEMiRCF.

2.3.3.1 Event Module

The Event module provides Event scheduling, cancellation, and execution services. Other modules that need to perform cyclic or timeout behaviors schedule Events to trigger their processing. The EventType enumeration lists all existing Events that may be scheduled so that they may be canceled. See Illustration 7 below.

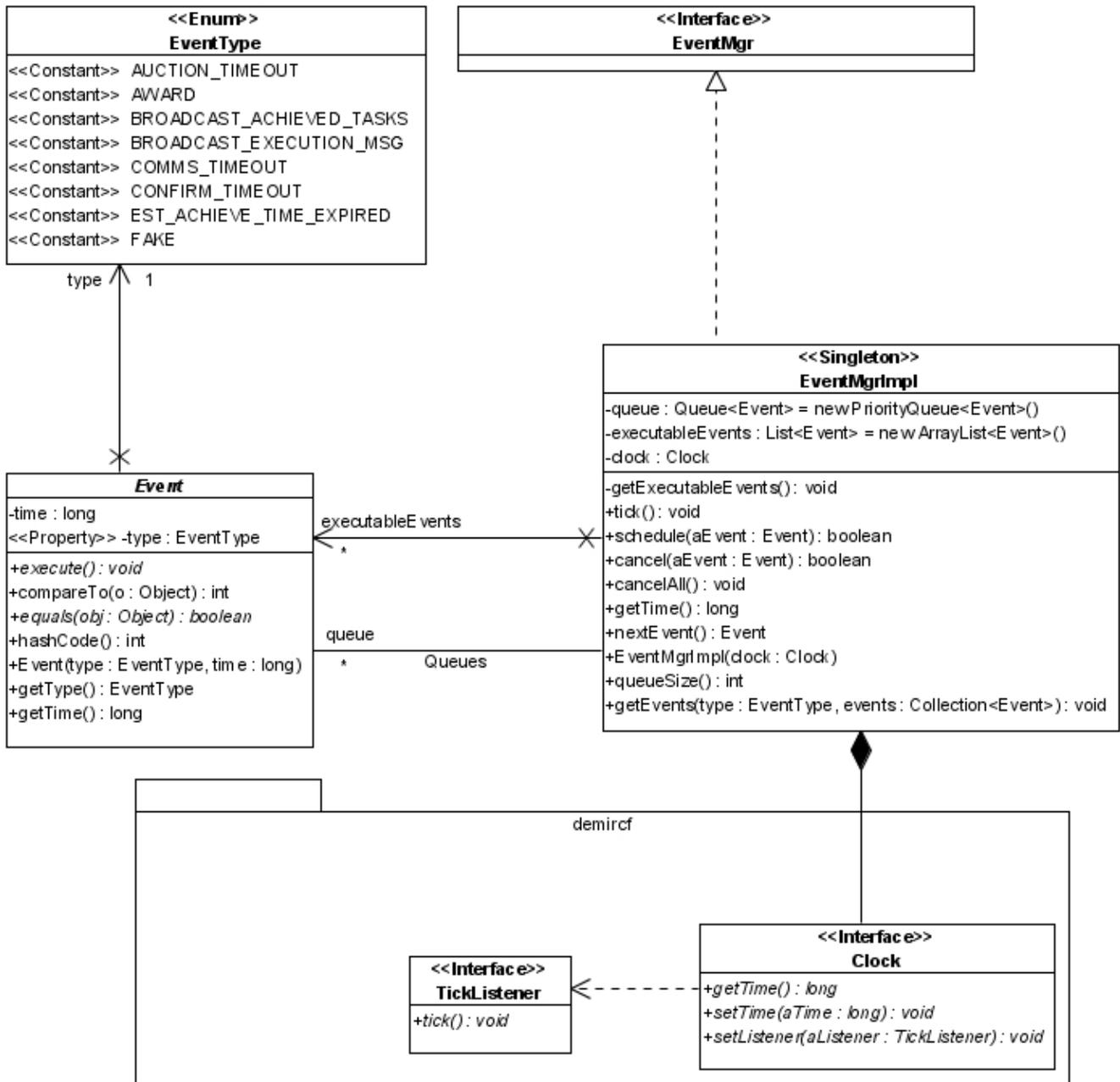


Illustration 7: DEMiR-CF Event Module

The `EventMgrImpl` class uses the `Clock` interface to trigger `Event` execution. I chose to use a `Clock` in an `Event` module because the notion of time step is prevalent in DEMiRCF literature.

2.3.3.2 Message Module

Illustration 8 below shows the Messages that are exchanged between DEMiRCF instances. Note that these are a subset of the messages described in Sariel's thesis [1] (p. 46). I discarded the message types that are needed for coalitions. I chose to include the Task in the MessageImpl base class because all Messages except MyLocationMsg needed a Task. MyLocationMsg sets the Task to null and clients should not access it. I included the sender name and capabilities to allow agent discovery to occur on receiving any Message.

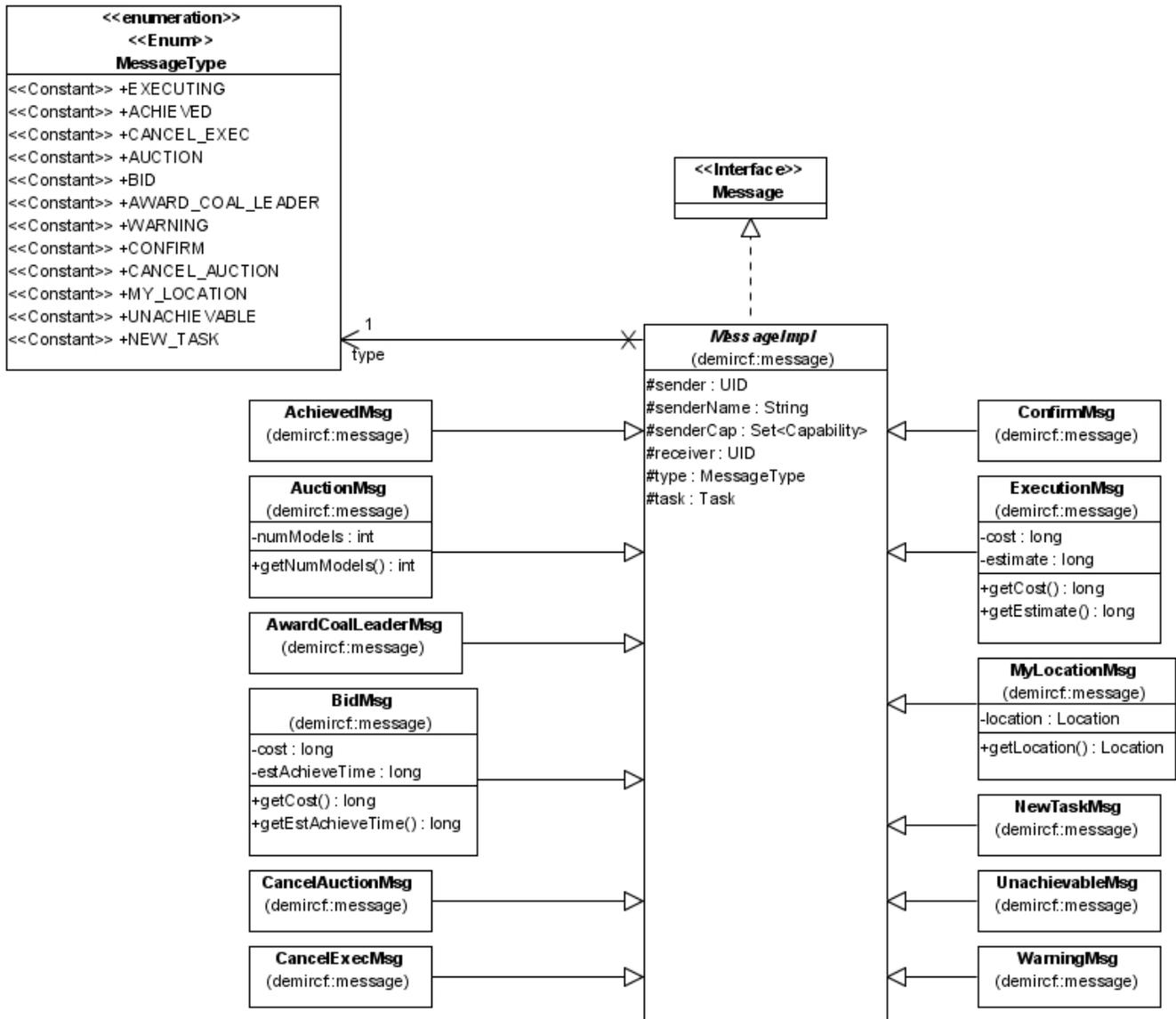


Illustration 8: DEMiR-CF Message Module

2.3.3.3 Model Module

Illustration 9 below shows the Model module Models.

The Model module represents an agent's world knowledge of Tasks and other Agents. I represent the

knowledge using finite state machines (FSMs) as Models per the thesis [1] (pp. 50-51). The Models define methods that act as event handlers to trigger state changes. Their base State class defines the same empty methods. Concrete States override the methods that cause them to transition the Model to a new current State. SingleAgentTaskModel extends TaskModel. When multiagent tasks are added, MultiAgentTaskModel should be added extending TaskModel.

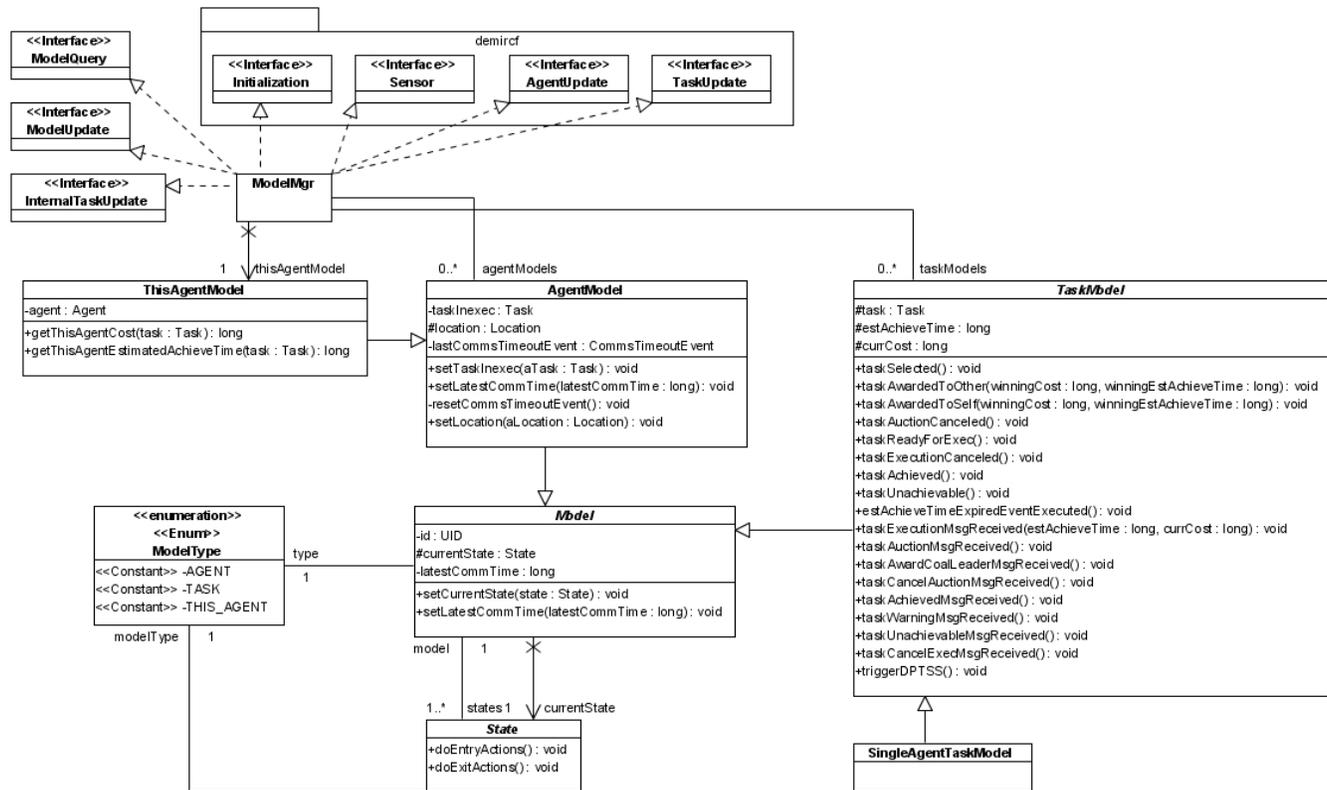


Illustration 9: DEMiR-CF Model Module Models

Illustration 10 below shows the Model module State classes.

AgentState extends State and is the base class for all concrete States an AgentModel may enter. TaskState extends State and is the base class for all concrete States a TaskModel/SingleAgentTaskModel may enter.

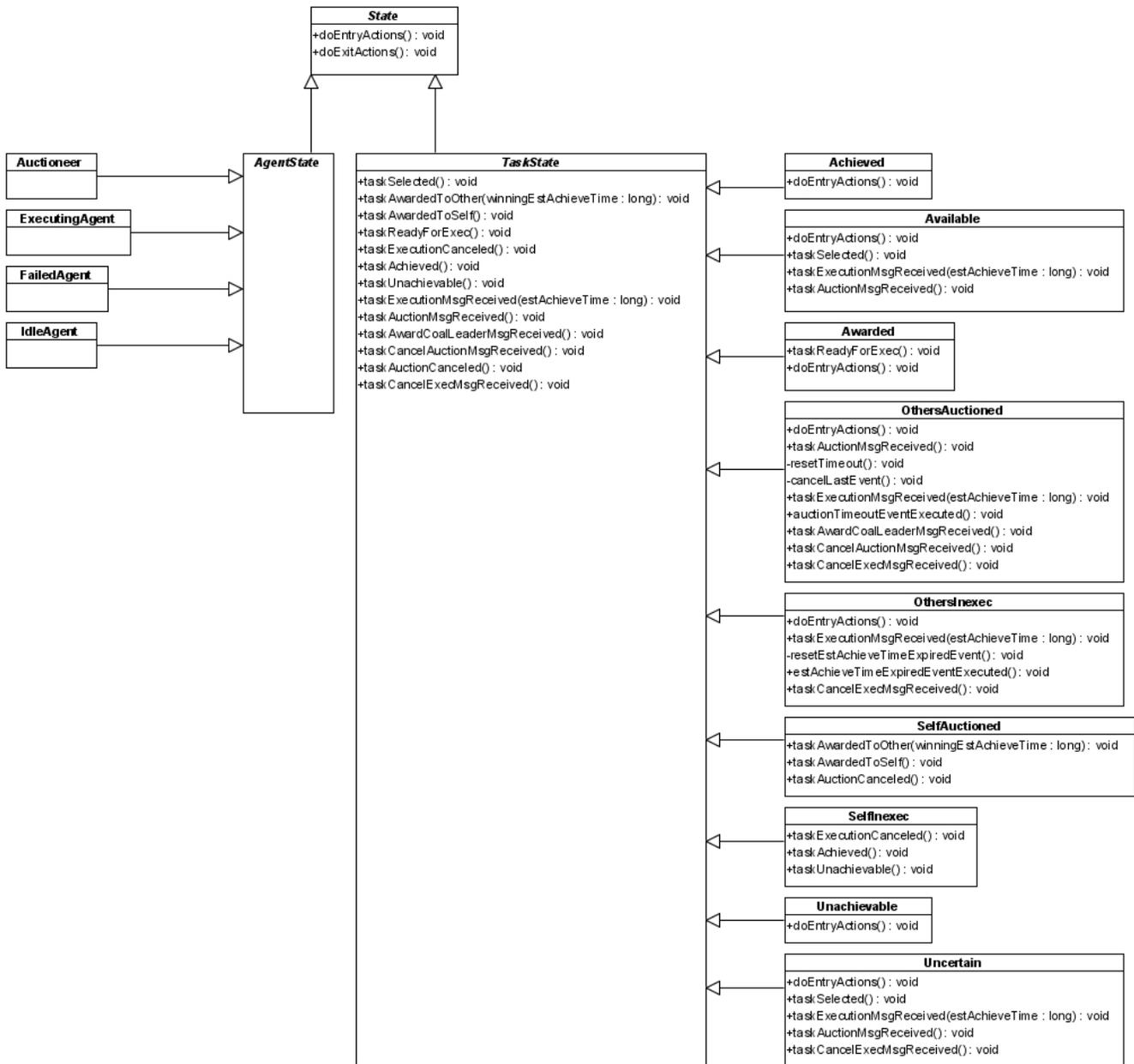


Illustration 10: DEMiR-CF Model Module States

Illustration 11 below shows the Events that the Model module uses for 1 cyclic and 2 timeout behaviors.

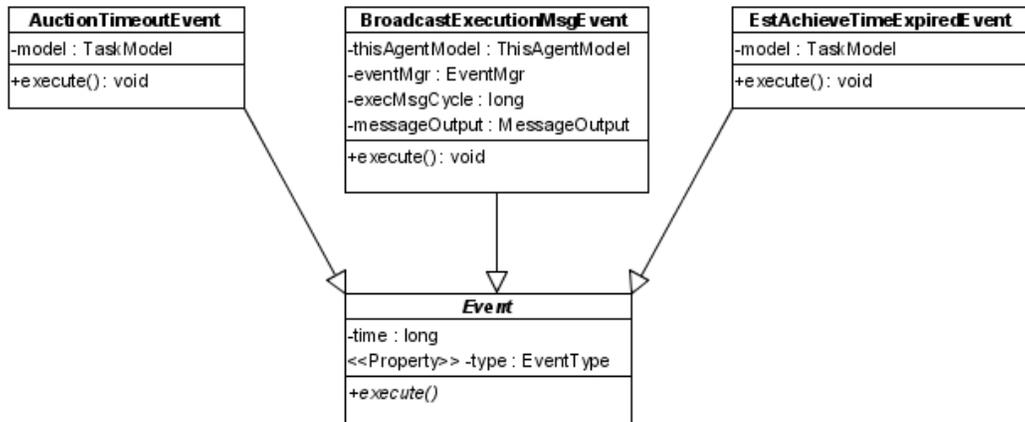


Illustration 11: DEMiR-CF Model Module Events

2.3.3.4 Allocation Module

Illustration 12 below shows the classes of the Allocation module.

The `AuctionMgrImpl` realizes the `AuctionMgr` interface and uses an `AuctionFactory` to create Auctions. The `AuctionImpl` implements the `Auction` interface.

The Allocation module has two timeout Events. The `AwardEvent` executes when bidding stops and the winner is awarded the Task. The `ConfirmTimeoutEvent` executes canceling the Auction if a `ConfirmMsg` is not received prior to its execution time.

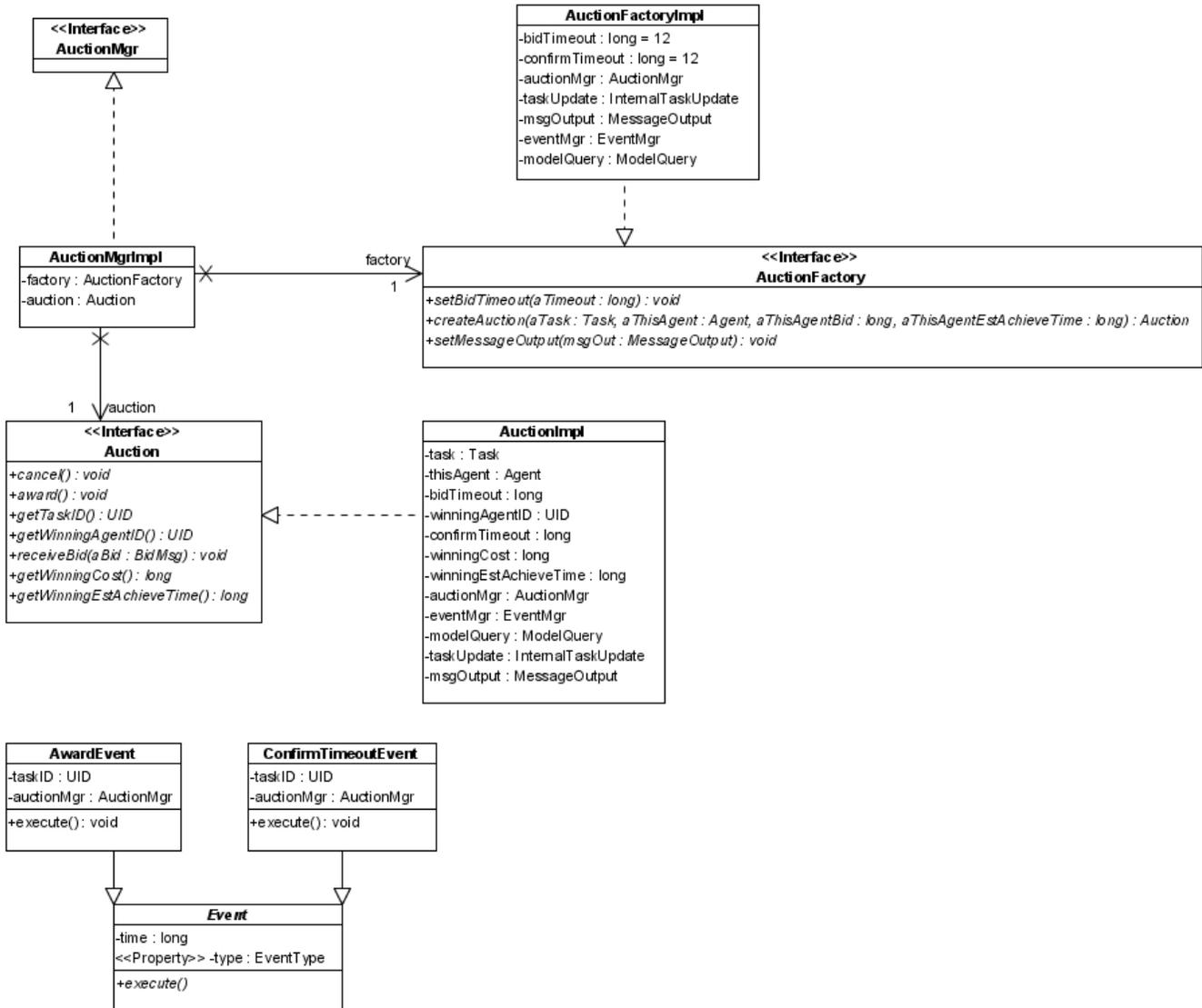


Illustration 12: DEMiR-CF Allocation Module

2.3.3.5 DPTSS Module

Illustration 13 below shows the classes in the DPTSS (Dynamic Priority-based Task Selection Scheme; Dynamic Task Selector) module.

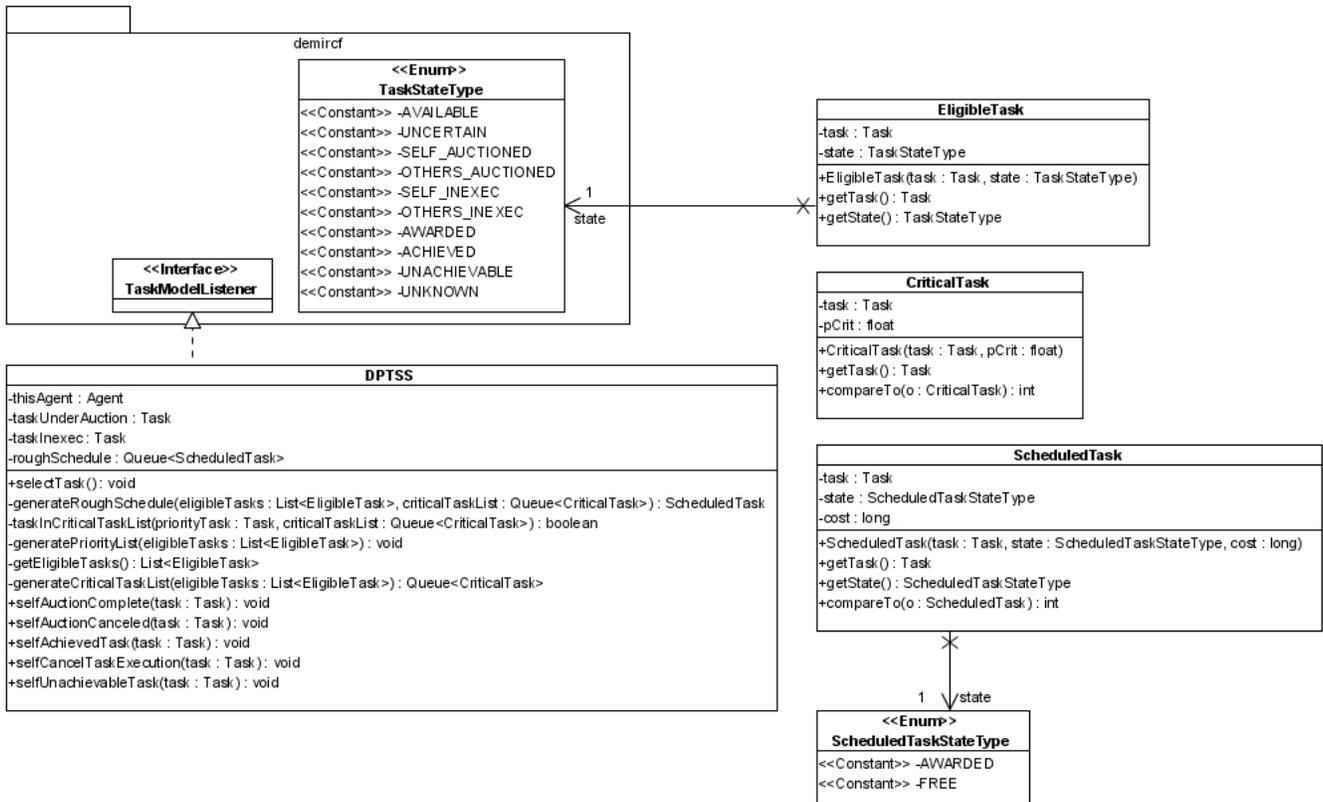


Illustration 13: DEMiR-CF DPTSS Module

This module implements the DPTSS algorithm in `selectTask` using the following steps: `getEligibleTasks`, `generateCriticalTaskList`, `generateRoughSchedule`, and `generatePriorityList`. The `EligibleTask` and `CriticalTask` are used in `getEligibleTasks` and `generateCriticalTaskList`, respectively, as one would expect. A `ScheduledTask` is used in `generatePriorityList` to encapsulate Tasks for sorting purposes. AWARDED Tasks are given priority over FREE Tasks. Within those categories, the minimum cost is used to sort the Tasks.

2.3.3.6 System Consistency Module

Illustration 14 below shows the interface and class in the Consistency module.

The `ModelUpdate` module uses the `check` method to determine if a received `Message` is consistent with current world knowledge. The `SystemConsistencyMgr` issues a `WarningMsg` if an inconsistency is found.

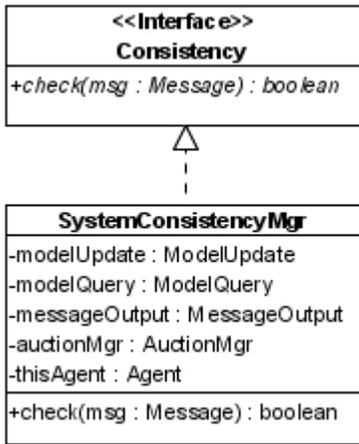


Illustration 14: DEMiR-CF System Consistency Module

2.3.3.7 Model Update Module

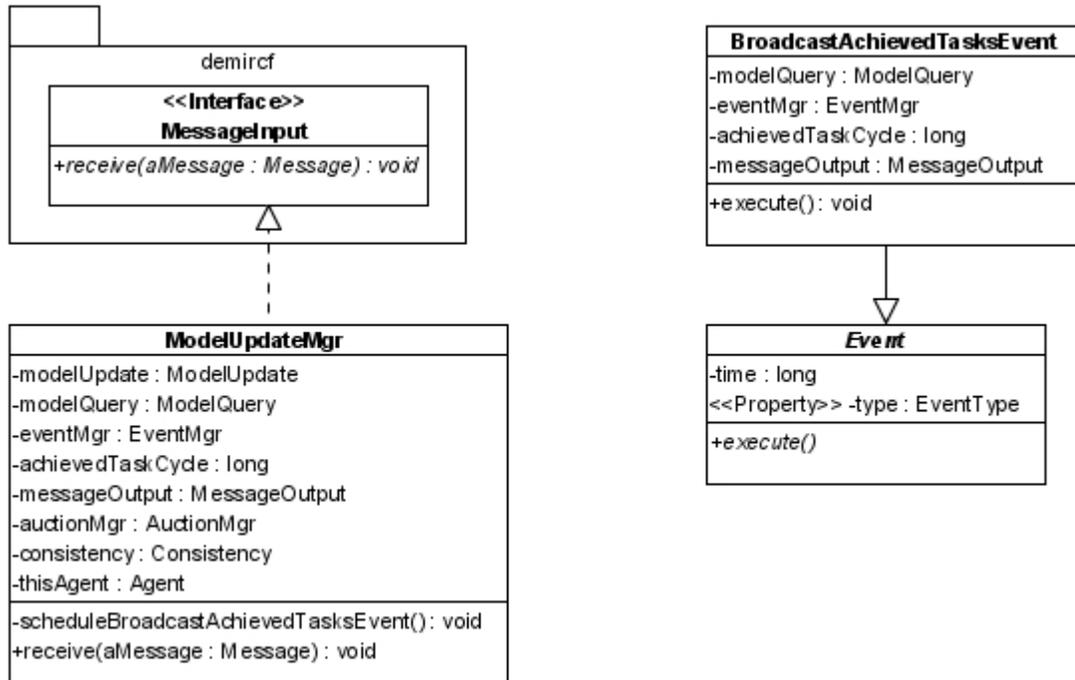


Illustration 15: DEMiR-CF Model Update Module

Illustration 15 above shows the interfaces and classes of the ModelUpdate module.

The ModelUpdateMgr invokes the Consistency module on all received Messages. If the Message is consistent with world knowledge it then relays the message to the Model module for update.

The BroadcastAchievedTasksEvent is a cyclic behavior implemented in the ModelUpdate module to notify all agents within broadcast range of known achieved Tasks using a bucket brigade approach. This repairs the world knowledge of agents that have been out of communication.

2.4 How to Use the Framework With OMACS

See 2.3.2.1 External Provided Interfaces and 2.3.2.2 External Required Interfaces for interfaces and their implementation requirements. This section provides additional details relevant to an application using GMoDS/OMACS.

Instance goal/role pairs and `demircf.Tasks` are corresponding equivalent concepts in GMoDS and DEMiR-CF. There must be a way to map from an instance goal/role pair to a `Task` and vice versa. Also, `demircf.CostFunctions` need access to the parameters of the instance goal to calculate the cost. Thus, the implementation of `demircf.Task` is a critical step in the integration process.

The `Agent` interface which extends the `TaskListener` interface is another critical implementation. This object must act as the conduit from DEMiR-CF to the execution component. The `TaskListener.changeTask` and `TaskListener.cancelTaskExecution` method calls must be relayed to the execution component to begin and stop execution of a given `Task`.

The `Agent.estimateAchievementTime` method call must be relayed to the execution component and from there to the plan executing for the `Task`. The plan is the object that knows the state of the `Task` and how many turns are likely needed to complete it. The `Agent.estimateAchievementTime` method will estimate an achievement time of 100 turns in the future for any `Task` not currently being executed by the `Agent` since I rely on the plan to have been initialized with the instance goal parameters to give the plan a state from which accurate estimates can be made. I chose this value to assure that a `Task` awarded to another `Agent` will not timeout of the `OthersInexec` state before the `Agent` executing the `Task` can provide an accurate estimate in its `ExecutionMsg`.

The control component must implement the `demircf.TaskUpdate` methods `taskReadyForExec`, `taskExecutionCanceled`, and `taskUnachievable` so that the execution component can inform the control component when it is ready to execute the assigned `Task` or when it has interrupted a `Task`. Each of these calls must be relayed to the corresponding DEMiRCF method.

`Agents` and `Tasks` must hold the `demircf.Capability` objects describing their owned and required capabilities, respectively. The `DEMiRCF.createCapability` method must be used to generate the `demircf.Capability` objects.

The control component must call `DEMiRCF.addTask` for any task that should not be auctioned and instead must be executed by every agent.

The control component must call `DEMiRCF.newTask` for any task that should be auctioned.

The control component must relay all `demircf.Messages` to the `DEMiRCF.receive` method. The control component should use the `AchievedMsg` to update the GMoDS goal model for `Tasks` achieved by other `Agents`. The control component must be able to handle multiple `AchievedMsgs` for the same `Task`.

The control component must call the `DEMiRCF.setLocation` method on each turn.

The control component must call `DEMiRCF.taskAchieved` each time it receives an `ACHIEVED` event.

If the application fails `Agents` it must call `DEMiRCF.agentFailed()`.

If the application recovers `Agents` it must call `DEMiRCF.agentRecovered(Task task)`.

An open question concerning `CostFunctions` is whether they need access to all `AgentModels` to access their locations to support a more global evaluation.

If GModS/OMACS supports tasks that require more than one agent to simultaneously execute them, the `MultiAgentTaskModel` must be implemented in `DEMiRCF`.

2.5 Framework Testing Overview

This section describes the types of unit tests built for DEMiR-CF and its component classes and the output that DEMiR-CF produces while operating.

2.5.1 Unit Tests

I developed JUnit tests using JUnit 3.8. The unit tests are found under DEMiR-CF-Share/src-tests. Production code is located in DEMiR-CF-Share/src-mike.

2.5.1.1 The demircf Package Unit Tests

`TestDEMiRCF` is the sole class defining test cases in the `demircf` package. All other classes are testing utilities.

`TestDEMiRCF` provides one test of the `createCapability` method and one long system level test of `DEMiRCF`.

2.5.1.2 The demircf.allocation Package Unit Tests

`TestAuctionMgrImpl` provides tests for creating and canceling an `Auction` and awarding a `Task` to this `Agent` or another `Agent`.

2.5.1.3 The demircf.consistency Package Unit Tests

`TestSystemConsistencyMgr` tests `Messages` that should generate a `WarningMsg`, `ExecutionMsgs` that should cause the receiving `Agent` to cancel or not cancel its own execution of the same `Task`, and `AuctionMsgs` that should cause the receiving `Agent` to cancel or not cancel its own `Auction`.

2.5.1.4 The demircf.dptss Package Unit Tests

`TestDPTSS` tests the method `selectTask` under various conditions: an `AWARDED` task is selected, a `FREE` task is selected, no eligible tasks remain, and when the current task under auction is re-selected.

2.5.1.5 The demircf.event Package Unit Tests

`TestEventManagerImpl` tests scheduling, canceling, and executing `Events`.

2.5.1.6 The demircf.model Package Unit Tests

`TestAgentModel` tests models of other `Agents` by setting the current state to all possible states.

`TestThisAgentModel` tests the model of the `Agent` client of `DEMiRCF` to assure that it will broadcast

ExecutionMsgs for the task currently in execution.

TestSingleAgentTaskModel tests the SingleAgentTaskModel to assure that it undergoes all desired transitions given the proper stimulus. In addition, this class tests that the SingleAgentTaskModel will trigger DPTSS upon appropriate transitions.

TestModelMgr tests the AgentModel, ThisAgentModel, and SingleAgentTaskModel as described above through the ModelMgr interface methods. In addition, it tests discovery of Tasks and Agents through receiving Messages, the effects of the AgentUpdate and TaskUpdate interfaces on the respective Models, and that receiving any Message should reset the communications timeout event.

2.5.1.7 The demircf.modelupdate Package Unit Tests

TestModelUpdateMgr tests that consistent and inconsistent Messages should update the latest communication time of the Agent and Task, ConfirmMsg and BidMsg should be relayed to the AuctionMgr, a BidMsg should be issued for every Task which the Agent is capable of executing, a ConfirmMsg is sent when an AwardCoalLeaderMsg is received, the BroadcastAchievedTasksEvent is executed, and that an AchievedTaskMsg for a Task in SelfInexec should cause a CancelExecMsg to be broadcast and the Agent.cancelTaskExecution method to be called.

2.5.2 Output

DEMiRCF uses the java.util.logging package to log its actions.

The DEMiRCF constructor establishes a log file as ~/Desktop/logs/YYYY.MM.DD/DEMiRCF-AgentName-HH-MM-SS.log. The logging level is set to FINE in the constructor.

Normal actions of DEMiRCF are logged at the FINE level. Errors are logged at the SEVERE level.

The "root" logger is named "demircf". Child loggers are named for the package in which they are used: "demircf.allocation", "demircf.consistency", "demircf.dptss", "demircf.model", and "demircf.modelupdate".

3 Object Construction Application

We selected the object construction application described in [2] (pp. 6-7) as a test bed for the DEMiR-CF framework implementation. Our application integrates GMoDS (Goal Model for Dynamic Systems) [4], OMACS (Organization Model for Adaptive Computational Systems) [5], and DEMiR-CF. GMoDS has specification goals to represent the task types and instance goals combined with OMACS roles to represent task instances that must be achieved to fulfill the application purpose. OMACS represents the roles, goals, and agents in an organization and the assignment of agents to roles that achieve goals. We use DEMiR-CF as a distributed algorithm to allocate tasks (instance goal/role pairs) to specific agents in an optimal manner. In this application, DEMiR-CF is given tasks that become active in GMoDS and thus are always active in DEMiR-CF. These tasks are only eligible, according to the DEMiR-CF definition, for the agents that have the required capabilities.

3.1 Object Construction Application Description

Illustration 16 below shows the object construction application main window. In this application, a group of agents cooperate to stack blocks in a column on top of a destination grid (gray grid in the picture). The idea is to stack all of the A blocks first, followed by B blocks, and finally C blocks. The agents must first find the blocks then push them to the proper locations. All agents have the capabilities (Sonar/BlobFinder) to find blocks. The

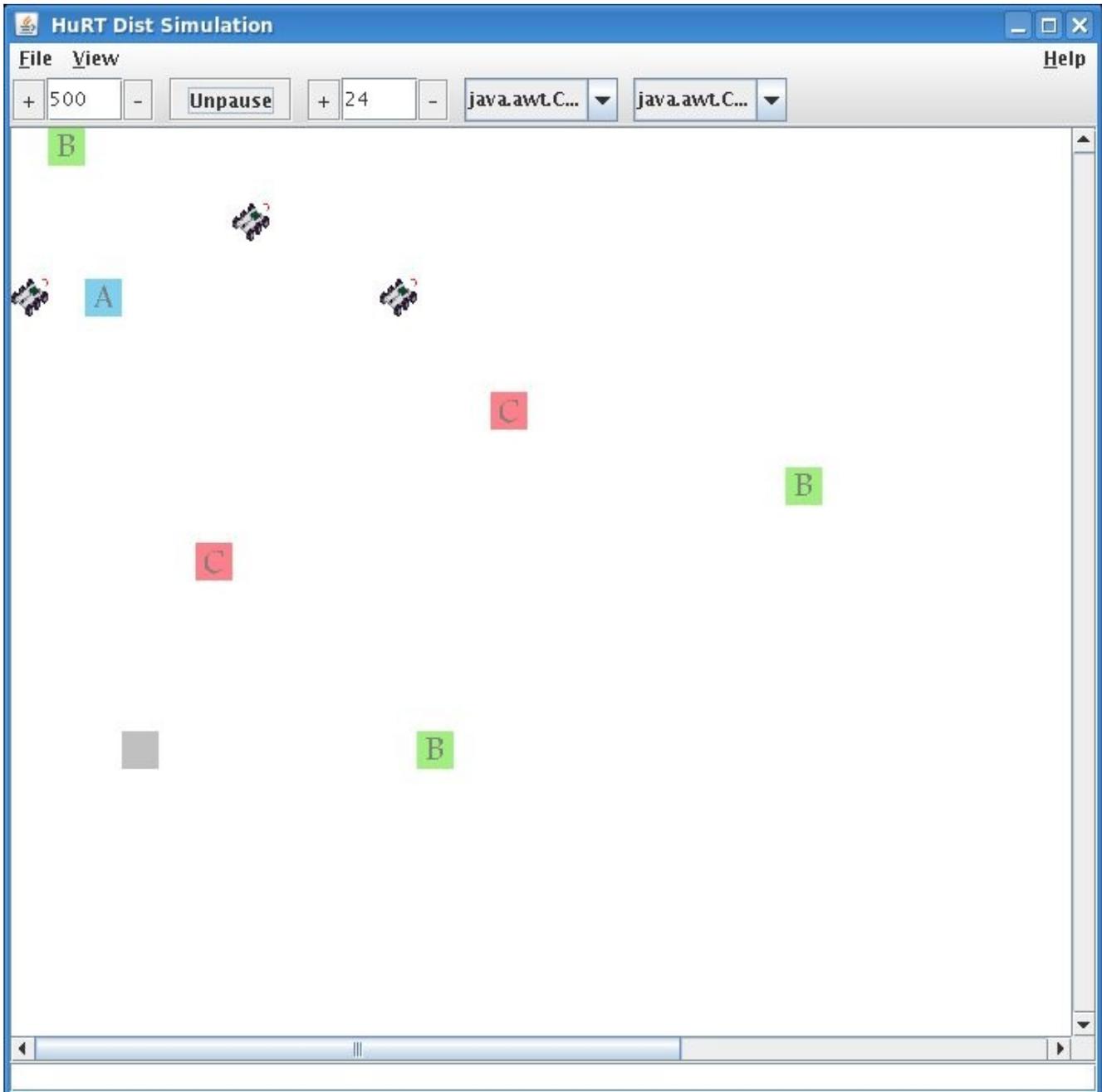


Illustration 16: Object Construction Application

SearcherAgent is limited to being able to find blocks. An agent type ACAgent can push A or C blocks using its ACProd. An agent type BAgent can push B blocks using its BProd.

3.1.1 Goal Model

The GModS goal model for the object construction application is shown in Illustration 17 below.

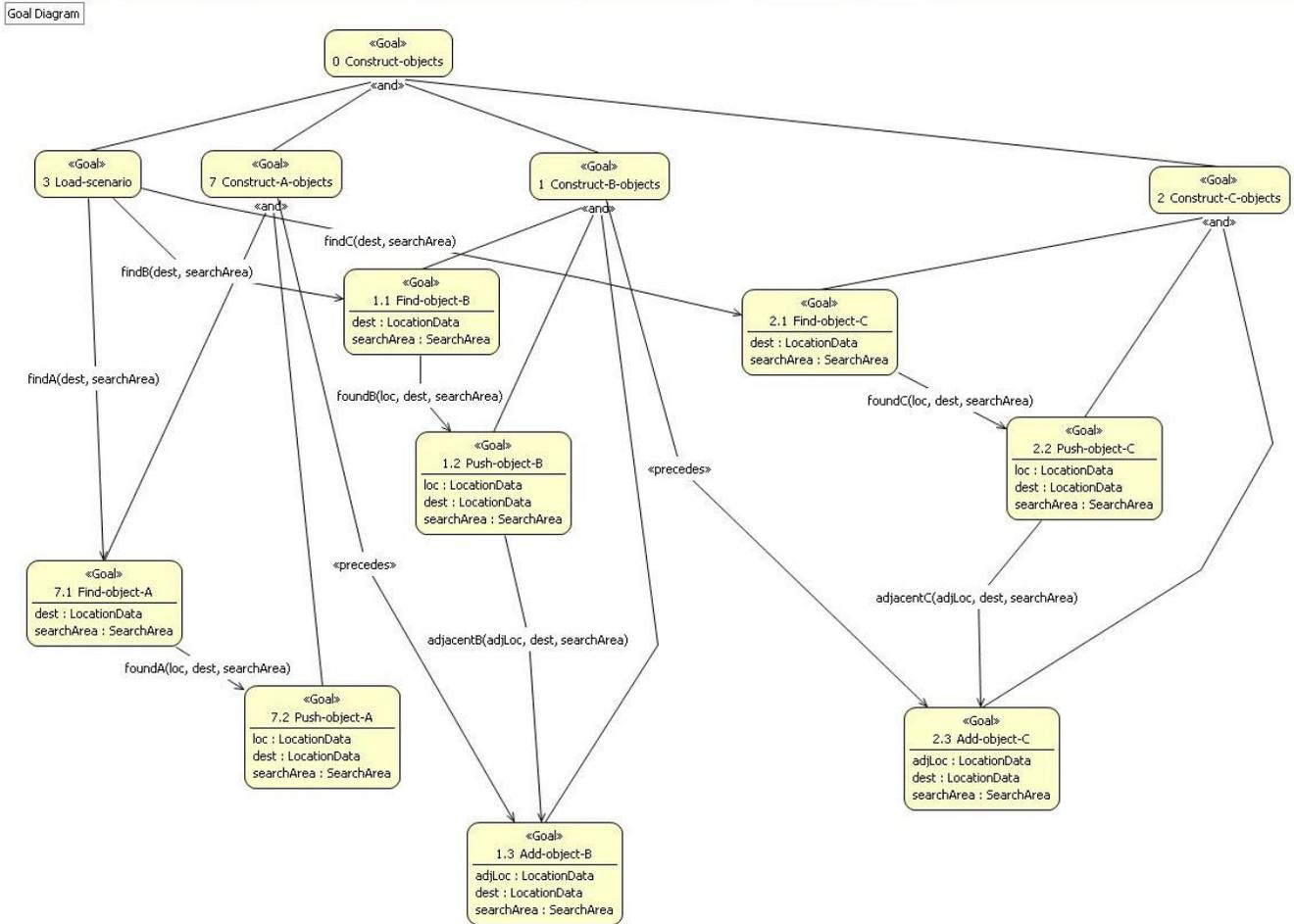


Illustration 17: Object Construction Application Goal Model

The goal specification tree shown in this figure has a top level goal of Construct-objects. Upon initialization the specification goal tree will instantiate one Load-scenario instance goal. This task is passed to `DEMiRCF.addTask`, a method which immediately passes it back to the `Agent.changeTask` method (this task is not auctioned; every agent must execute it). The Load-scenario goal triggers one Find-object-A goal for every object A, one Find-object-B goal for every object B, and one Find-object-C goal for every object C in the environment.

When an agent executing Find-object-A finds an A the event `foundA` triggers a Push-object-A goal. The agent executing Push-object-A goes to the location of object A, grabs it, pushes it to the top of the building site, drops it, and moves out of the way.

When an agent executing Find-object-B finds a B the event `foundB` triggers a Push-object-B goal. The agent executing Push-object-B goes to the location of object B, grabs it, and pushes it to the top of a tower of B objects adjacent to the building site where the object B is dropped. When the object B is dropped, the event `adjacentB` is executed triggering a Add-object-B goal. This goal is not active until the goal Construct-A-objects is achieved (when all object A's are pushed to the top of the building). When the Add-object-B goals can be

activated, the agent executing this goal goes to the location adjacent to the building, grabs the object B, and pushes it to the top of the building site. When all object B's are at the top of the building, Construct-B-objects is achieved.

When an agent executing Find-object-C finds a C the event foundC triggers a Push-object-C goal. The agent executing Push-object-C goes to the location of object C, grabs it, and pushes it to the top of a tower of C objects adjacent to the building site where the object C is dropped. When the object C is dropped, the event adjacentC is executed triggering a Add-object-C goal. This goal is not active until the goal Construct-B-objects is achieved (when all object B's are pushed to the top of the building). When the Add-object-C goals can be activated, the agent executing this goal goes to the location adjacent to the building, grabs the object C, and pushes it to the top of the building site. When all object C's are at the top of the building, Construct-C-objects is achieved. When Construct-C-objects is achieved the overall Construct-objects goal is achieved and the system halts.

3.1.2 Role Model

The OMACS role model for the object construction application is shown in Illustration 18 below.

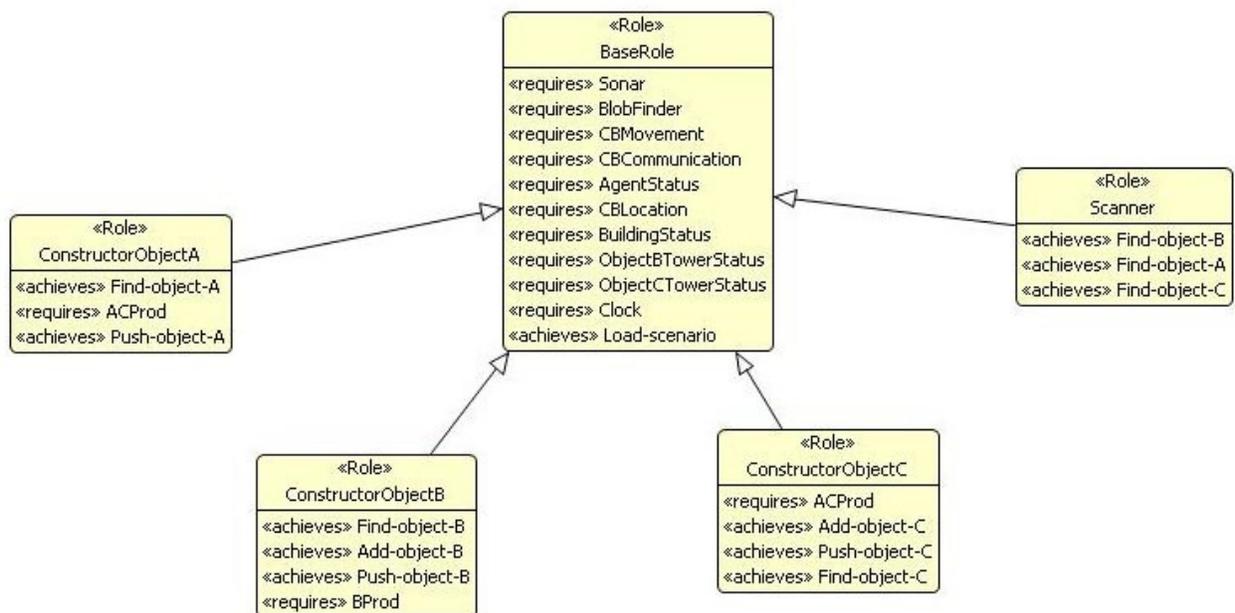


Illustration 18: Object Construction Application Role Model

The BaseRole requires the Sonar, BlobFinder, CBMovement, CBCommunication, AgentStatus, CBLocation, BuildingStatus, ObjectBTowerStatus, ObjectCTowerStatus, and Clock capabilities. The BaseRole achieves the Load-scenario goal.

The Scanner role extends the BaseRole and achieves the Find-object-A, Find-object-B, and Find-object-C goals.

The ConstructorObjectA role extends the BaseRole, requires the ACProd capability and achieves the Find-object-A and Push-object-A goals.

The ConstructorObjectB role extends the BaseRole, requires the BProd capability and achieves the Find-object-B, Push-object-B, and Add-object-B goals.

The ConstructorObjectC role extends the BaseRole, requires the ACProd capability and achieves the Find-object-C, Push-object-C, and Add-object-C goals.

3.2 Execution Component Design Overview

This section describes the components I designed and implemented for the object construction application except where noted.

3.2.1 Capabilities

Illustration 19 below shows the sensor capabilities.

The BlobFinder.identify method provides the ObjectType of a TangibleObject at the given absolute coordinates and if the ObjectType matches the desiredType the object is marked as identified (preventing other agents from identifying it again).

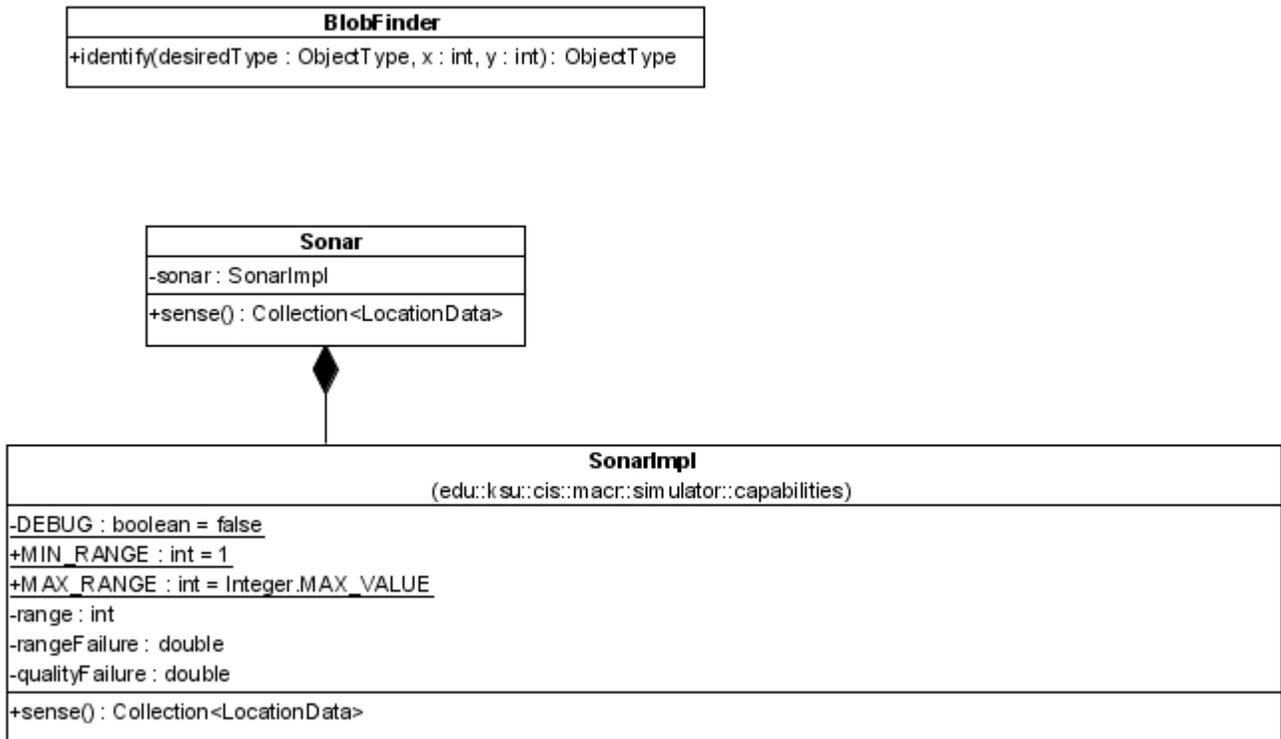


Illustration 19: Object Construction Application Sensor Capabilities

The Sonar capability uses the implementation provided in the edu.ksu.cis.macr.simulator.capabilities package.

Illustration 20 below shows the Prod capabilities. Every Prod can grab or drop a TangibleObject at a specified absolute location and can be queried for the attached object. The ProdImpl can be configured to allow attachment of only certain ObjectTypes; the ACProd allows only object A or C to be grabbed. The BProd allows only object B to be grabbed.

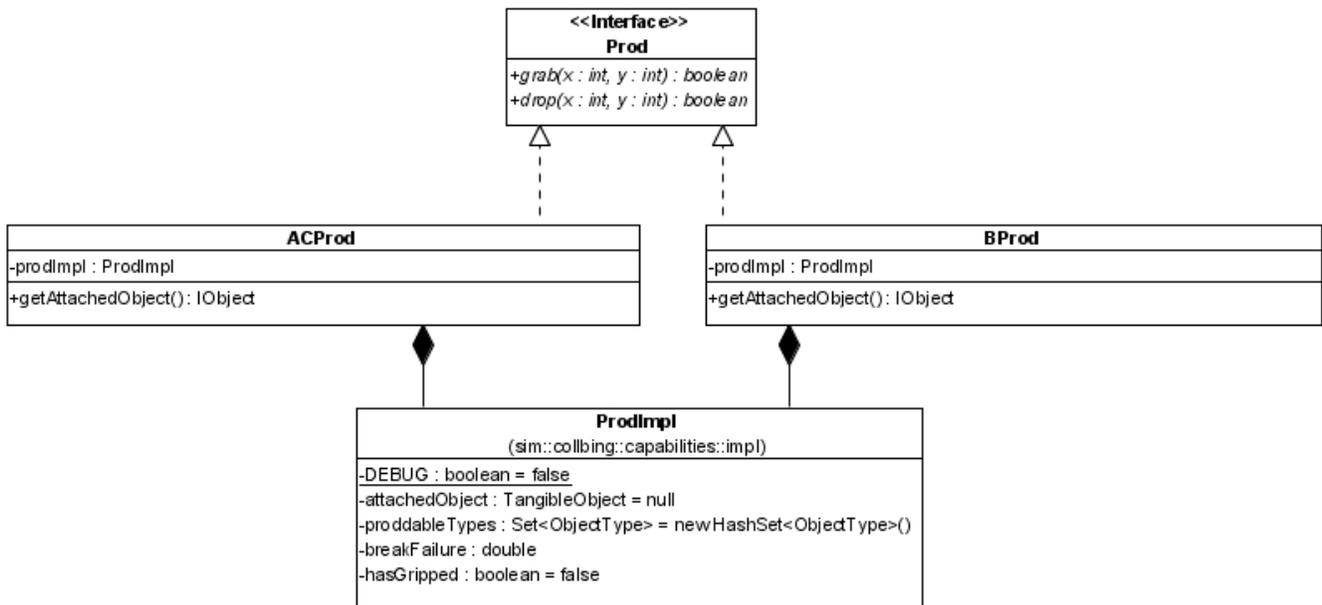


Illustration 20: Object Construction Application Prod Capabilities

Illustration 21 below shows the status capabilities. Agents use these capabilities to track and exchange knowledge of the current top occupied grid in the building or object B or C towers.

The `BuildingStatus` capability tracks the status of the building.

The `ObjectBTowerStatus` capability tracks the status of the adjacent object B tower.

The `ObjectCTowerStatus` capability tracks the status of the adjacent object C tower.

The `Clock` capability is used to trigger DEMiR-CF events based on the number of elapsed turns for the agent.

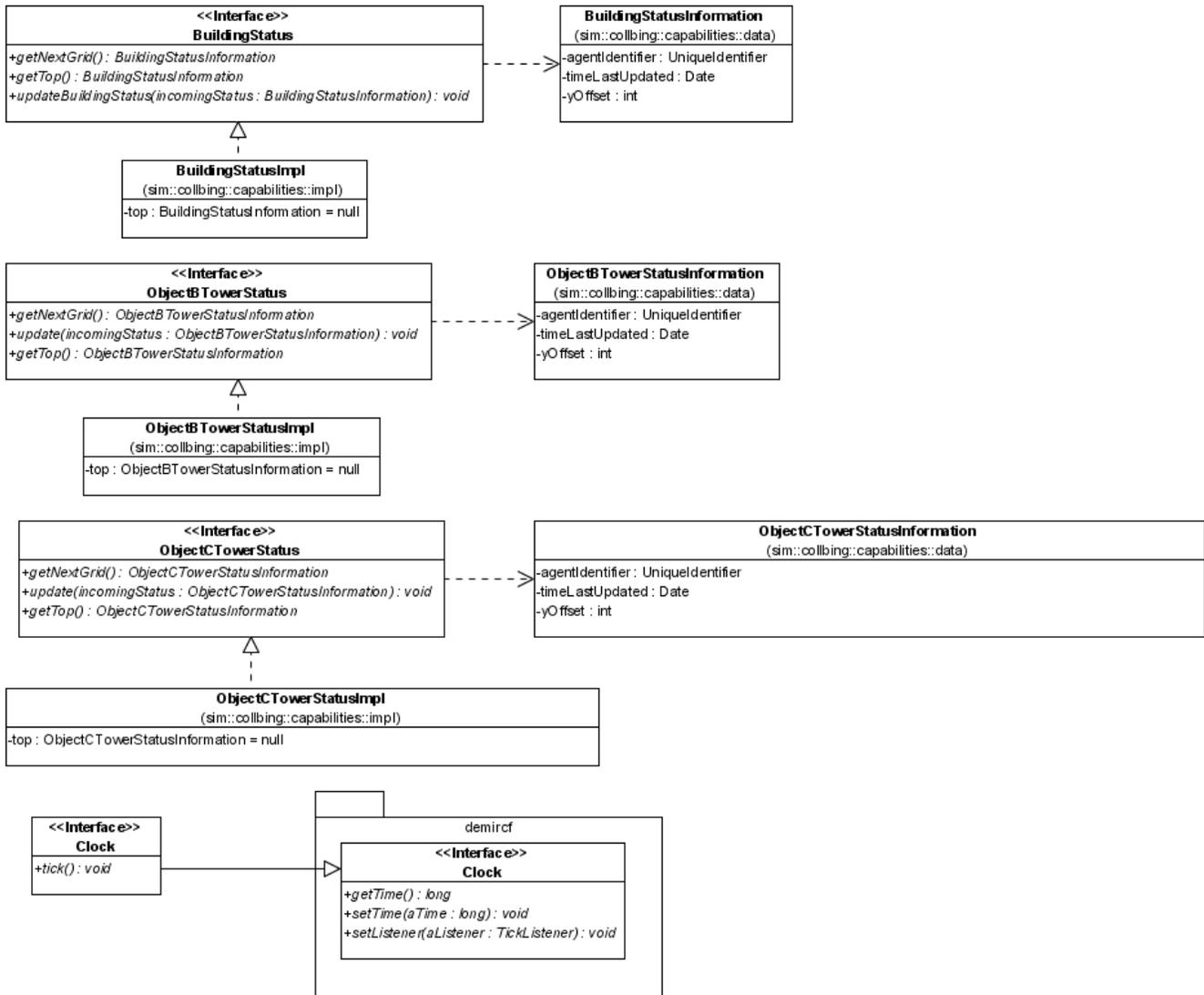


Illustration 21: Object Construction Application Status Capabilities

Illustration 22 below shows capabilities developed by Jorge Valenzuela.

The CBLocation capability is used to give the agent access to its location.

The CBMovement capability allows the agent holonomic movement.

The AgentStatus provides status needed by OMACS.

The CBCommunication capability provides communication between agents.

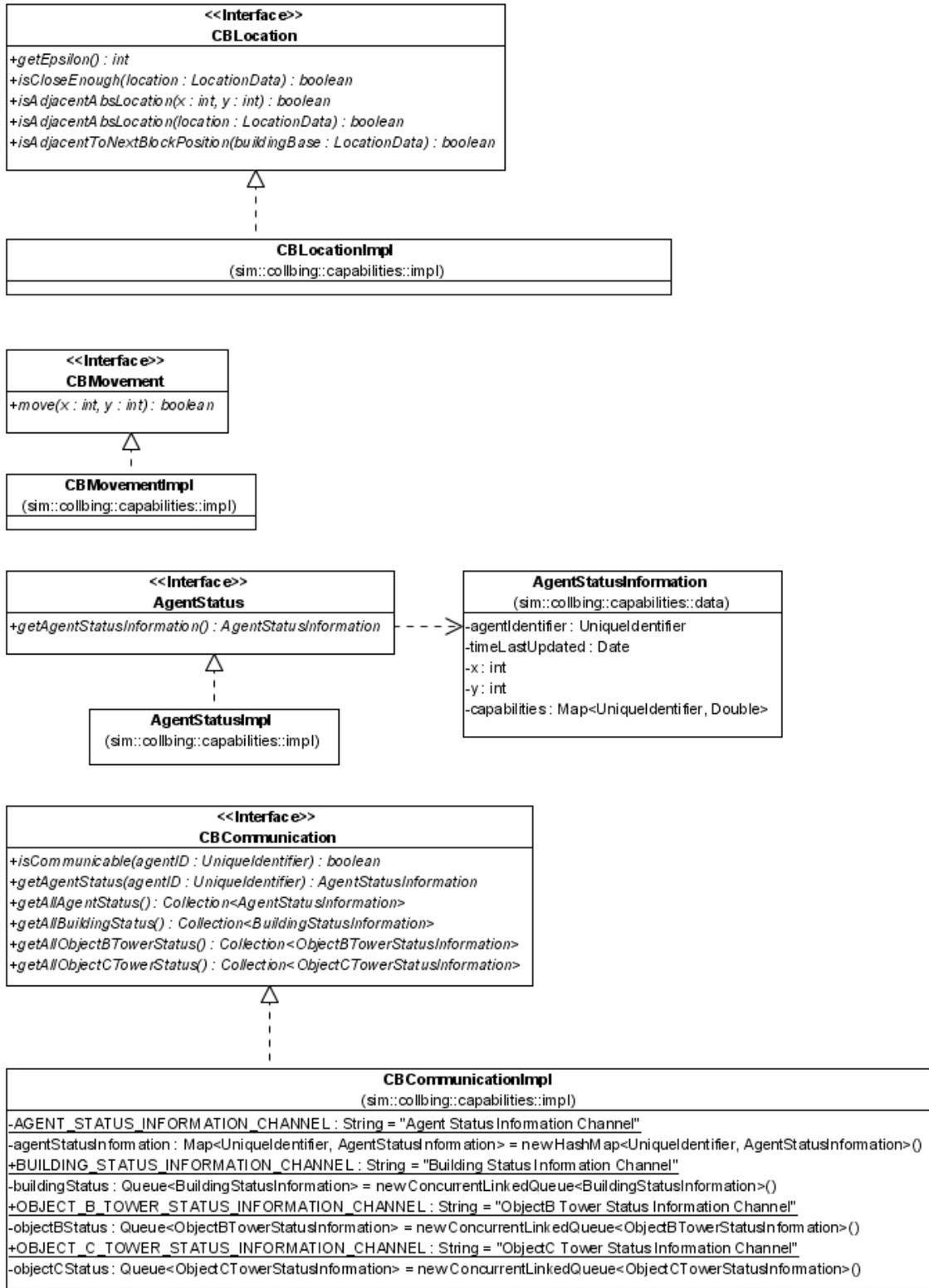


Illustration 22: Object Construction Application Other Capabilities

3.2.2 Plans

This section describes the Plan objects in the object construction application. All of the Plan objects implement the `CBEExecutionPlan` interface. All of the "PlanCode" classes shown below extend the `AbstractPlanCode` class (not shown due to space limitations) and therefore have access to the building location and the search area.

Illustration 23 below shows the `LoadScenarioPlan`. This plan achieves the Load-scenario goal when its `execute` method is called, executing the triggers described in 3.1.1 above. It accesses the objects in the environment to generate the proper number of Find-object-X goals of each type passing each goal the building site "destination" and a `SearchArea` that contains all objects.

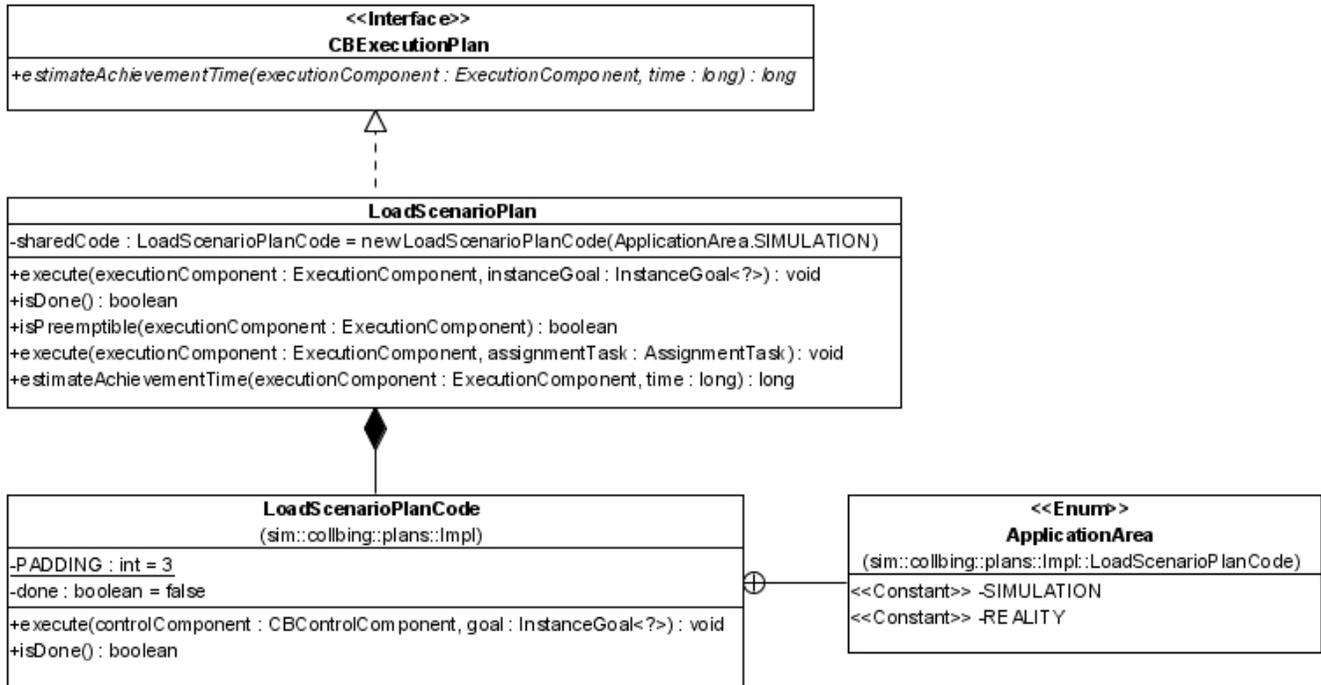


Illustration 23: Object Construction Application LoadScenarioPlan

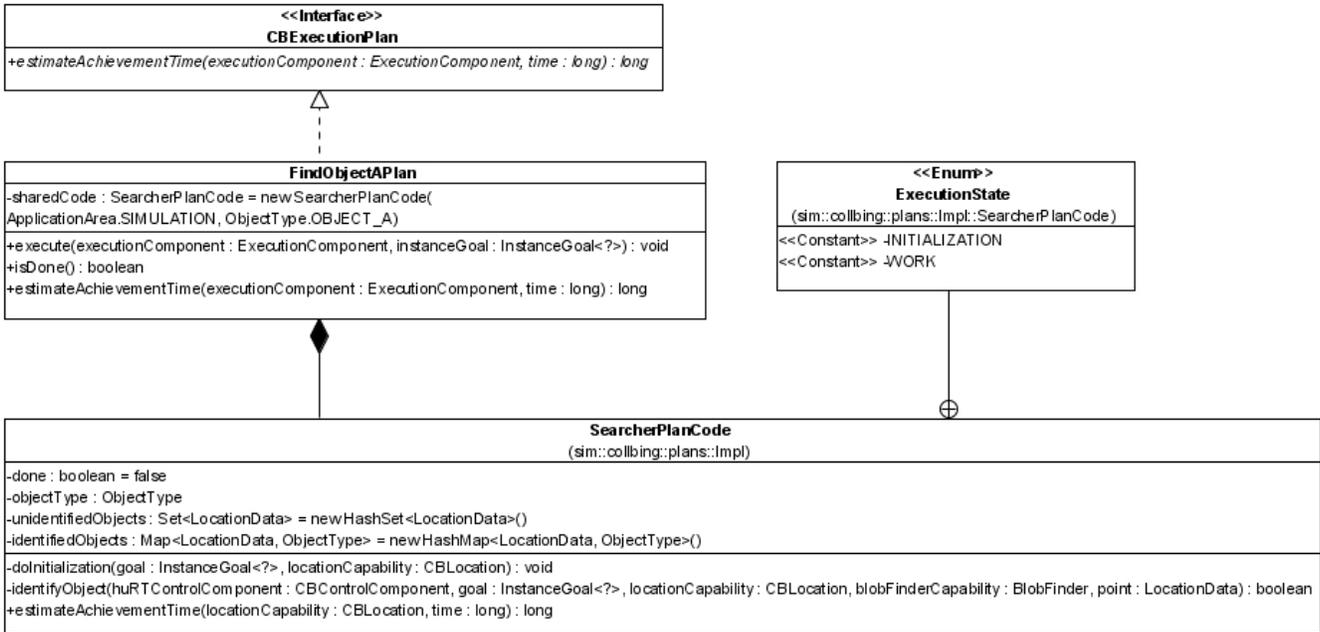


Illustration 24: Object Construction Application FindObjectAPlan

Illustration 24 above shows the FindObjectAPlan. This plan achieves the Find-object-A goal when its execute method is called, executing the triggers described in 3.1.1 above. SearcherPlanCode implements the execute method keeping track of the ObjectType being looked for, and identified and unidentified objects that have been found. Agents executing this plan move to random locations within the SearchArea and identify all objects matching the desired type.

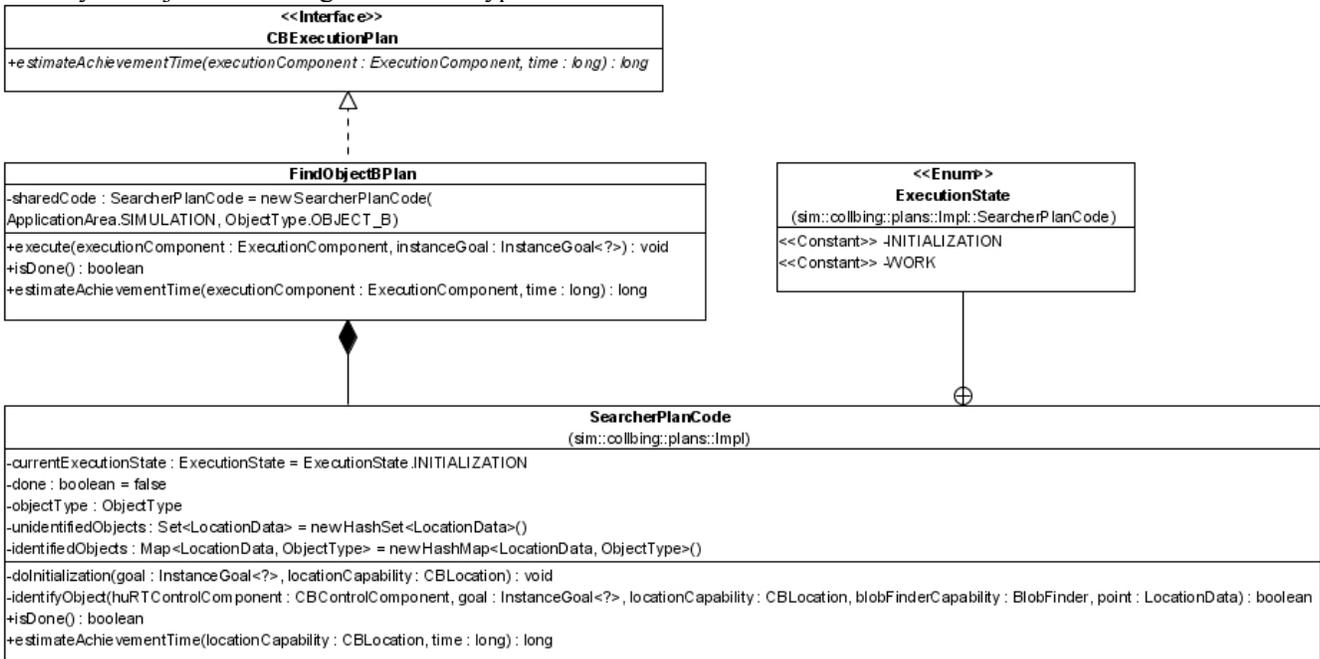


Illustration 25: Object Construction Application FindObjectBPlan

Illustration 25 above shows the FindObjectBPlan. Its design matches FindObjectAPlan except for the type of object desired.

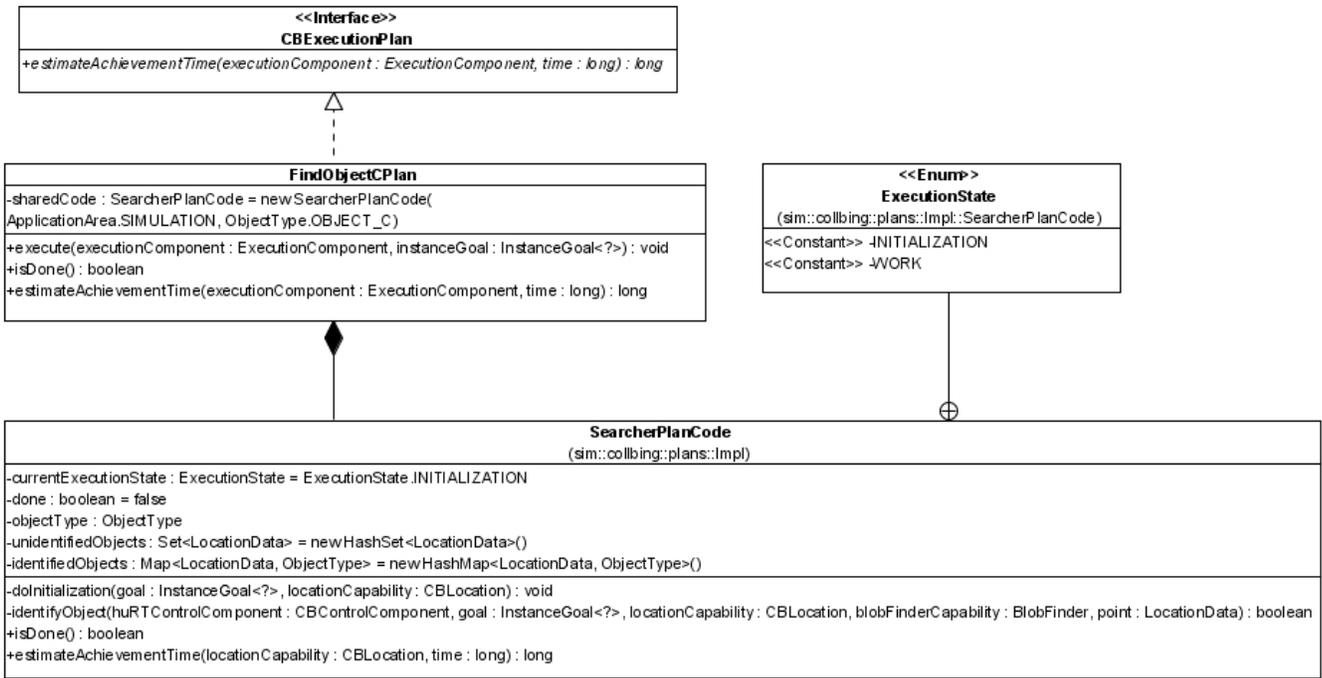


Illustration 26: Object Construction Application FindObjectCPlan

Illustration 26 above shows the FindObjectCPlan. Its design matches FindObjectAPlan except for the type of object desired.

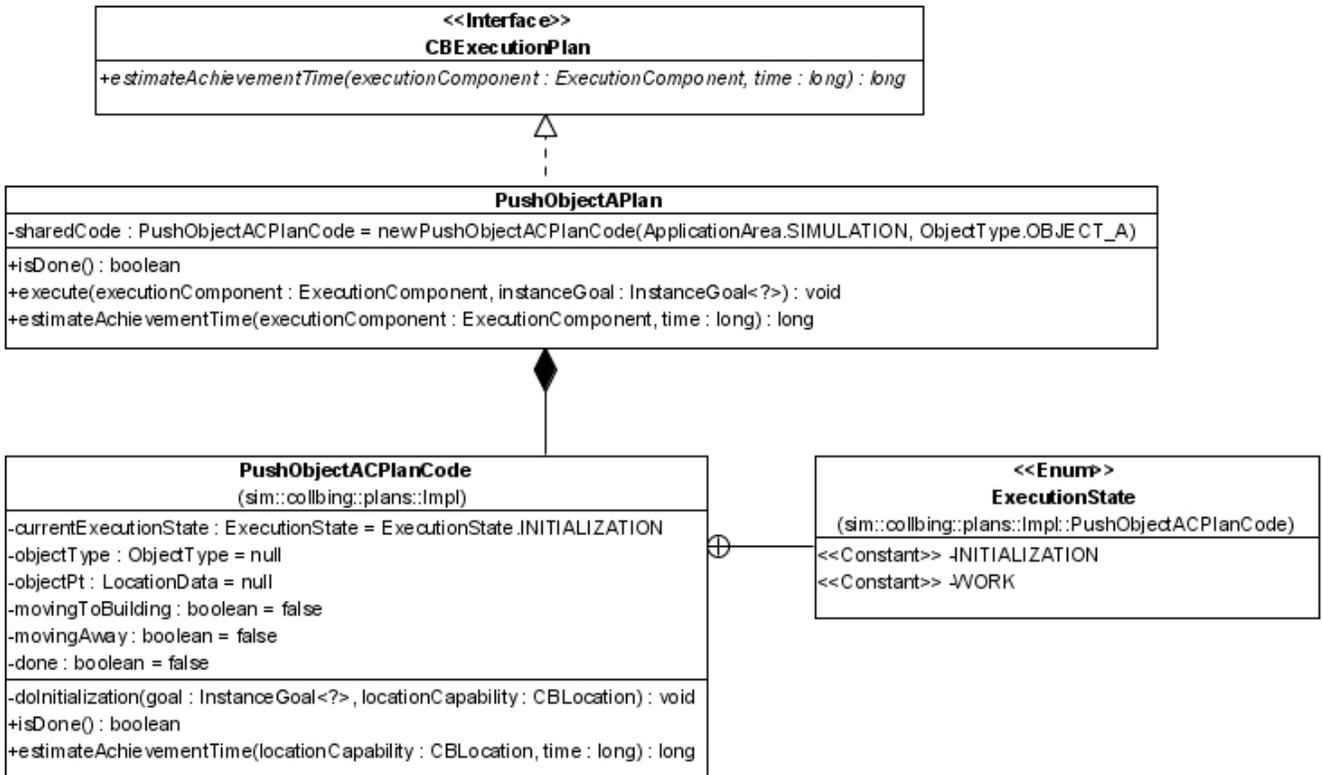


Illustration 27: Object Construction Application PushObjectAPlan

Illustration 27 above shows the `PushObjectAPlan`. This plan achieves the Push-object-A goal when its execute method is called, executing the triggers described in 3.1.1 above. `PushObjectAPlanCode` is configured with the desired object type A and implements the execute method. `PushObjectAPlanCode` tracks the desired object type, the location to pick up the object, whether the agent is moving to the building with the grabbed object, and whether it is moving away from the building after dropping the object. The object is pushed to the current top of the building site and dropped.

Illustration 28 below shows the `PushObjectBPlan`. This plan achieves the Push-object-B goal when its execute method is called, executing the triggers described in 3.1.1 above. `PushObjectBPlanCode` implements the execute method. `PushObjectBPlanCode` tracks the location to pick up the object, the object B tower location, whether the agent is moving to the object B tower with the grabbed object, and whether it is moving away from the tower after dropping the object. The object is pushed to the current top of the object B tower and dropped.

Illustration 29 below shows the `PushObjectCPlan`. This plan achieves the Push-object-C goal when its execute method is called, executing the triggers described in 3.1.1 above. `PushObjectCPlanCode` implements the execute method. `PushObjectCPlanCode` tracks the location to pick up the object, the object C tower location, whether the agent is moving to the object C tower with the grabbed object, and whether it is moving away from the tower after dropping the object. The object is pushed to the current top of the object C tower and dropped.

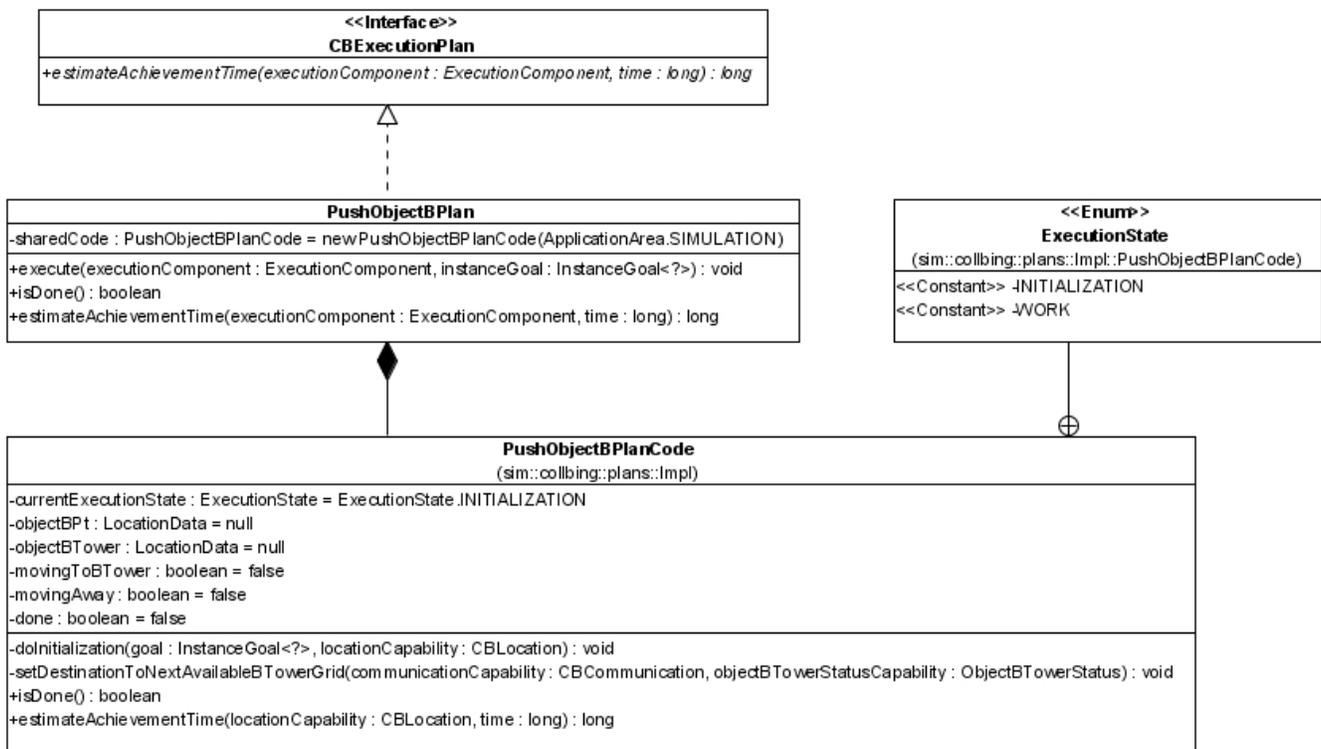


Illustration 28: Object Construction Application `PushObjectBPlan`

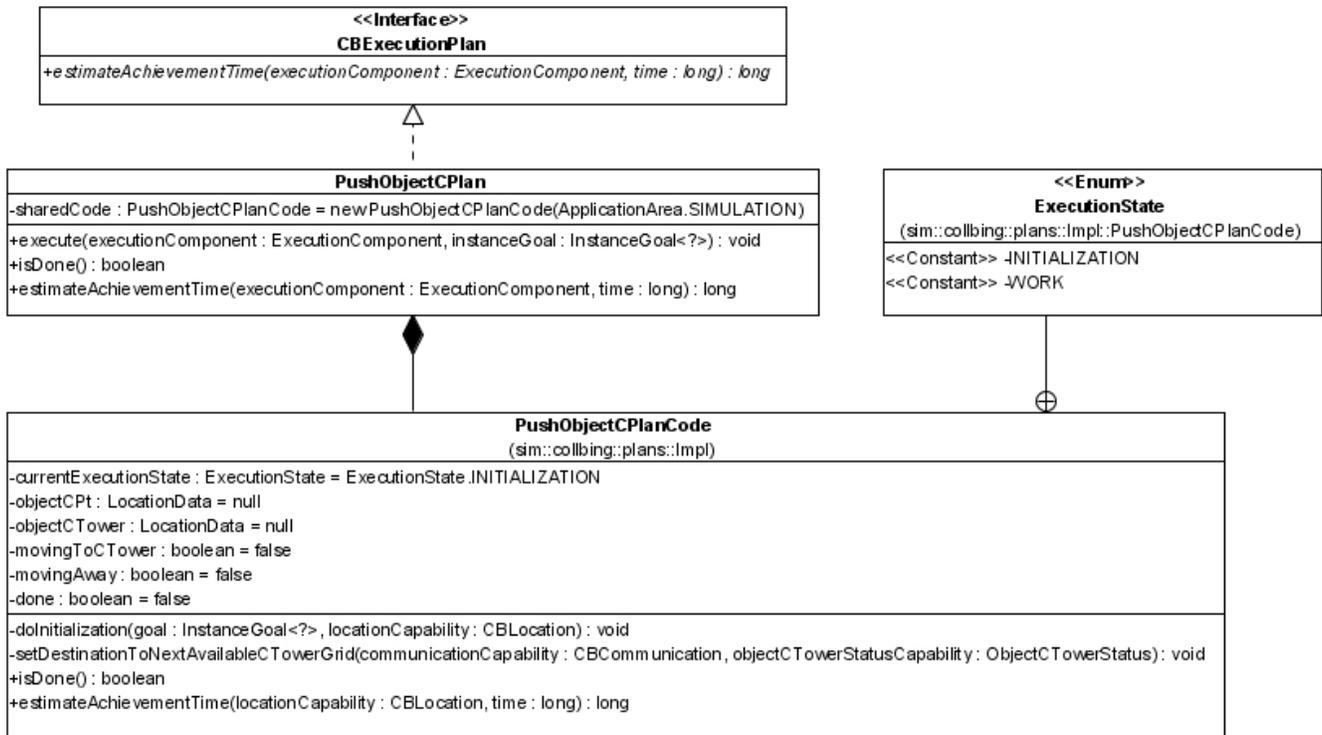


Illustration 29: Object Construction Application PushObjectCPlan

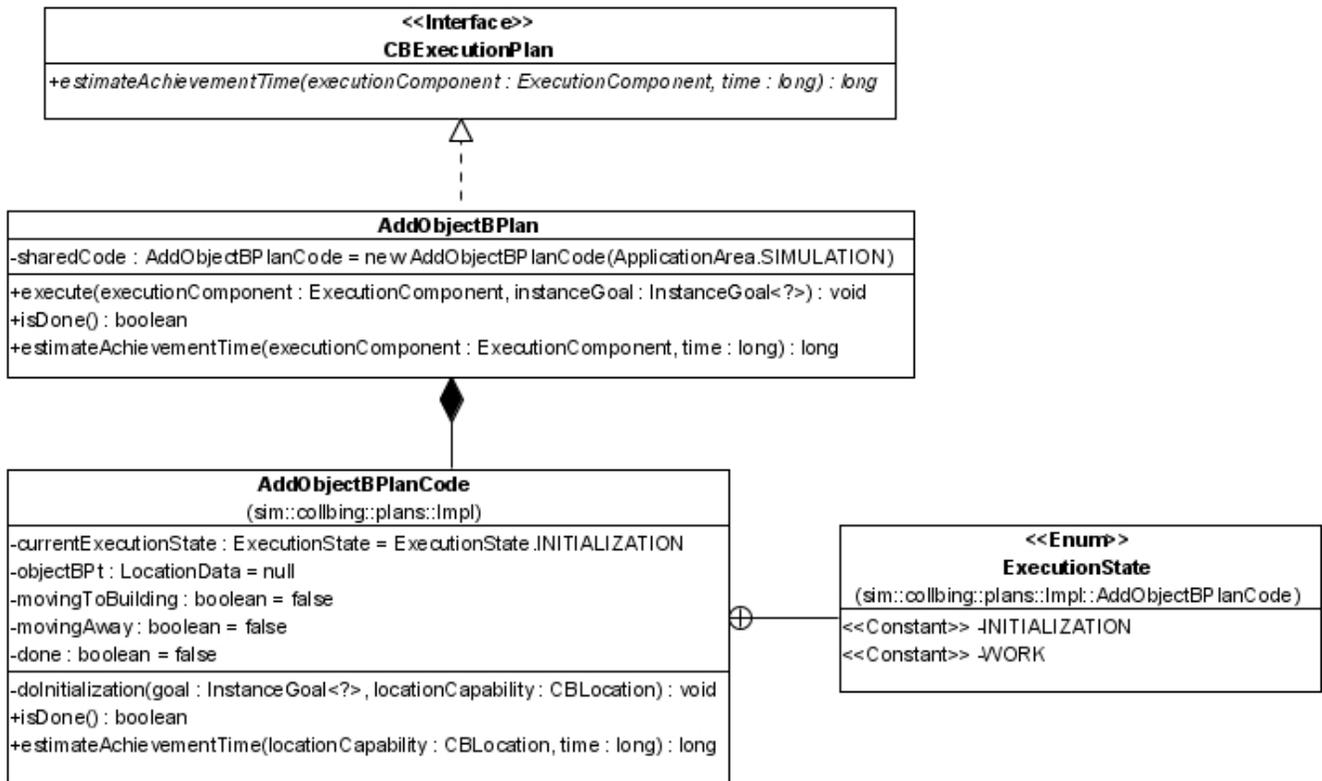


Illustration 30: Object Construction Application AddObjectBPlan

Illustration 30 above shows the `AddObjectBPlan`. This plan achieves the Add-object-B goal when its execute method is called. `AddObjectBPlanCode` implements the execute method. `AddObjectBPlanCode` tracks the location to pick up the object, whether the agent is moving to the building with the grabbed object, and whether it is moving away from the building after dropping the object. The object is pushed to the current top of the building and dropped.

Illustration 31 below shows the `AddObjectCPlan`. This plan achieves the Add-object-C goal when its execute method is called. `PushObjectACPlanCode` is configured with the desired object type C and implements the execute method. `PushObjectACPlanCode` tracks the desired object type, the location to pick up the object, whether the agent is moving to the building with the grabbed object, and whether it is moving away from the building after dropping the object. The object is pushed to the current top of the building site and dropped.

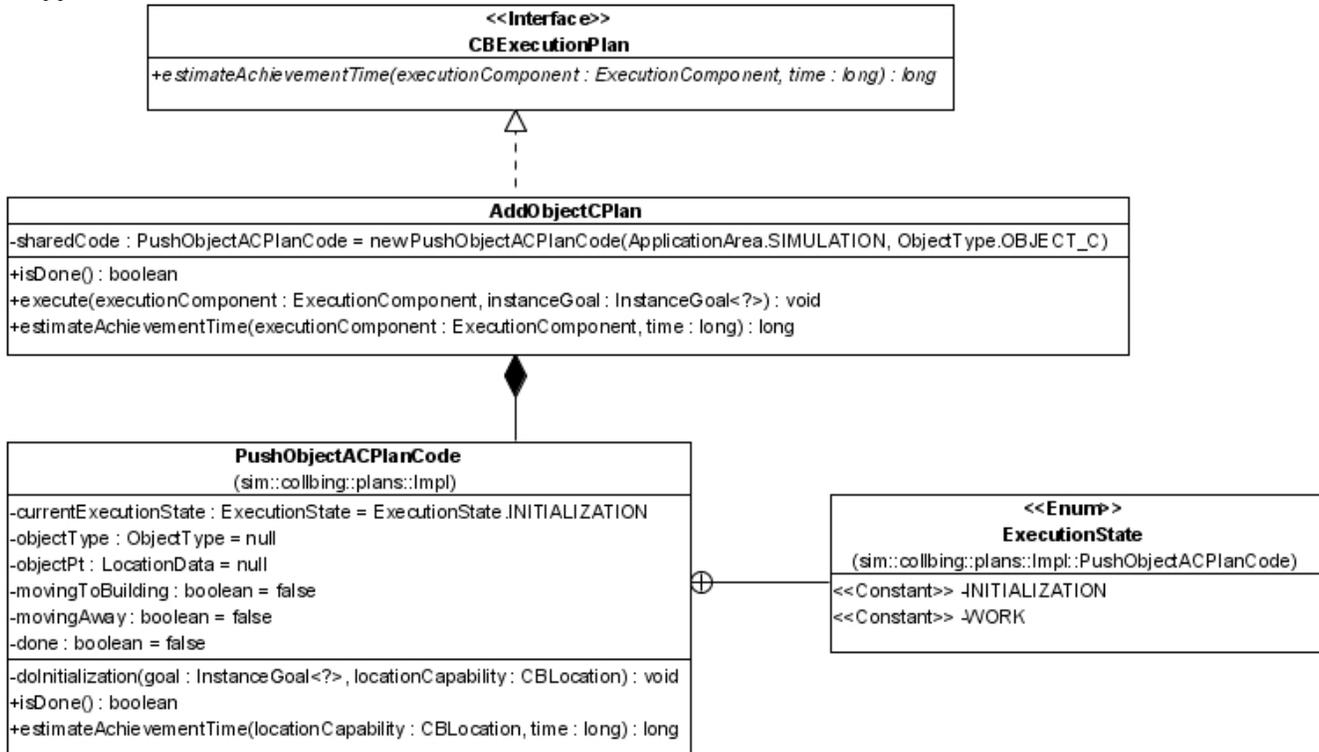


Illustration 31: Object Construction Application `AddObjectCPlan`

3.2.3 Agents

This section describes the Agent classes.

Illustration 32 below shows the Agent classes.

`BasicCBAgent` is the base class for all concrete Agent classes. The `BasicCBAgent.robotCode` method was originally implemented by Jorge Valenzuela. I extended it to allow testing of the Plan code without relying on the control component to issue `AssignmentTasks` and made other minor adjustments. To test the Plans, set `BasicCBAgent.TEST_PLANS` to true and `AbstractPlanCode.TEST` to true and recompile. The `getTestPlan` method is overridden by concrete `BasicCBAgents` to provide the plan to be tested for that agent.

The `BasicCBAgent.endTurn()` extends the `AbstractAgent.endTurn()` method to allow the

clock capability to function. I had to remove the "final" qualifier from `AbstractAgent.endTurn()` to allow `BasicCBAgent` to compile.

`BasicCBAgent` has an attribute of type `DEMiRCFConfig` to set the `DEMiRCF` parameter values that control the cyclic and timeout behavior intervals.

The `auctionTimeout` is the timeout threshold for an `Auction` initiated by another `Agent` to result in a valid `ExecutionMsg`. If violated, the `Task` enters the `Uncertain` state. The suggested value for `auctionTimeout` is 25. The `auctionTimeout` should be greater than or equal to `bidTimeout + confirmTimeout + execMsgCycle` because these times represent processing steps that must occur for an auction to result in a valid `ExecutionMsg`.

The `bidTimeout` is the timeout threshold for other `Agents` to bid in an `Auction` initiated by this `Agent`. At the end of this period, this `Agent` awards the `Task` to the lowest bidder. The suggested value for `bidTimeout` is 10.

The `confirmTimeout` is the timeout for `ConfirmMsg` reception from another `Agent` awarded a `Task` by the auctioneer. If no `ConfirmMsg` is received from the awarded `Agent` in this time period, the `Auction` is canceled. The suggested value of `confirmTimeout` is 5.

The `achievedTaskCycle` is the time cycle between broadcasts of known achieved `Tasks` in a bucket brigade style. The suggested value of `achievedTaskCycle` is 15. The `achievedTaskCycle` can be greater than `execMsgCycle` since the intent is to help repair knowledge of agents that have lost communications and not prove that this agent hasn't failed.

The `execMsgCycle` is the time cycle between broadcasts of `ExecutionMsgs`. This message is intended to keep `Agents` aware of the current cost for a `Task`, the current estimated achievement time, and prove that the sender has not failed. The suggested value of `execMsgCycle` is 5.

The `commsTimeout` is the timeout for communications. If no `Message` is received by `Agent A` from another `Agent B` within this time period, then `Agent A` assumes `Agent B` has failed and that the `Task` it was executing is now uncertain. The suggested value of `commsTimeout` is 10. The `commsTimeout` should be greater than `execMsgCycle`.

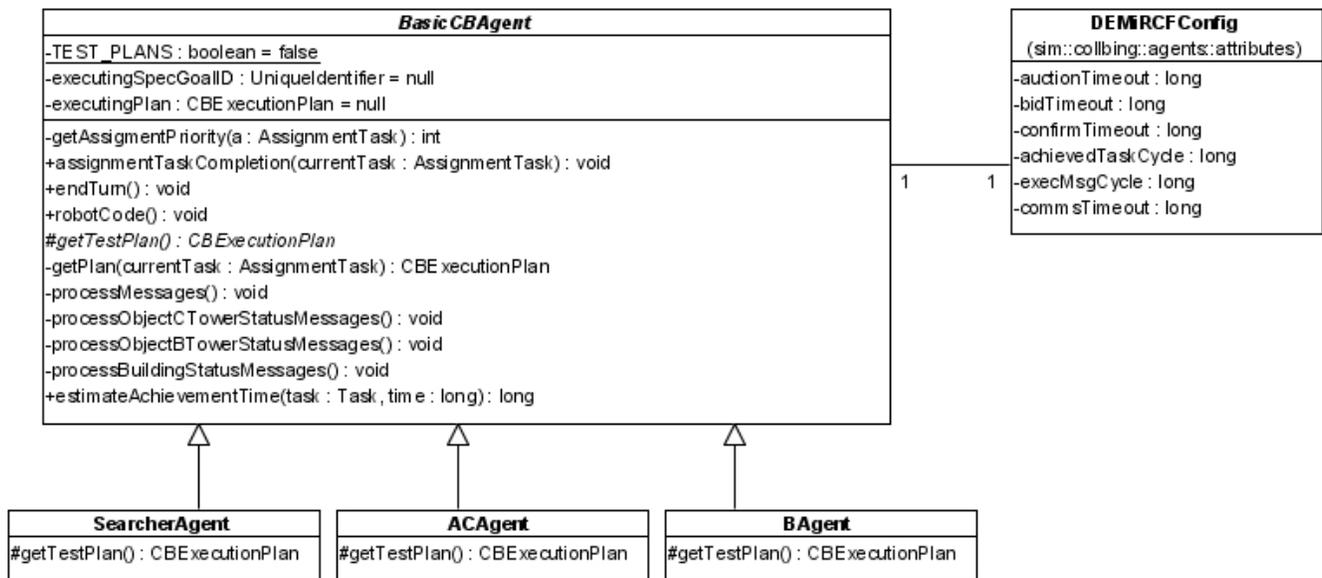


Illustration 32: Object Construction Application Agents

3.2.4 Cost Functions

The application must provide specific `demircf.CostFunction` implementations that optimize the task allocation. Illustration 33 below shows the application-specific `demircf.CostFunctions` and utility classes used to map them to the correct `demircf.Task`.

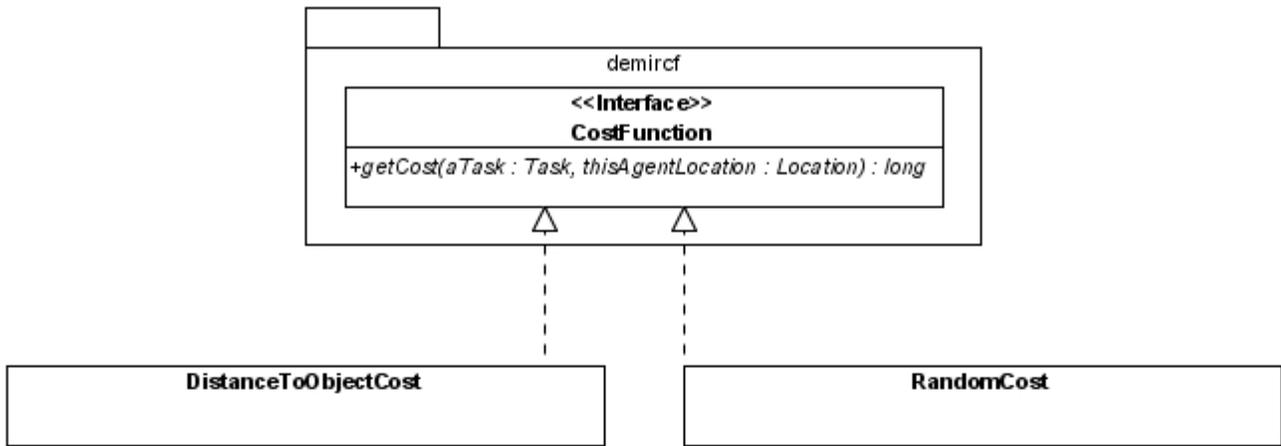
The `DistanceToObjectCost` is used for all `demircf.Tasks` except `Load-scenario` which is not auctioned and the `Find-object-X` tasks that use `RandomCost`. `DistanceToObjectCost` measures the distance from the agent to the object pickup location.

The `RandomCost` is used for the searching `Tasks` for simplicity's sake.

Note that `CostFunction.getCost` may need additional parameters to support other types of implementations. For example, to give the `SearcherAgent` preference for the search tasks over the agents with `Prods` one will need to add the `Agent` interface as a parameter and endow that interface with additional methods that can distinguish the agent types. As mentioned in 2.4 How to Use the Framework With OMACS, another example of additional parameters is that the `CostFunction.getCost` method may need the other `AgentModels` known by `DEMiRCF` to access their reported locations for a global optimization function that attempts to minimize the total path length traveled by all agents.

The `CostFunctionProvider` maps the name of the specification goal (`String`) to the `CostFunction` that should be used for the corresponding `demircf.Task`.

The `CostFunctionXMLUtils` and `CostFunctionClassUtils` are used when reading the XML configuration file that maps each `demircf.Task` to its `CostFunction`.



CostFunctionProvider
-mapping : Map<String, CostFunction> = newHashMap<String, CostFunction>()
+addCostFunction(specificationGoalID : String, costFunction : CostFunction) : void
+getCostFunction(specificationGoalID : String) : CostFunction

CostFunctionXMLUtils
+ELEMENT_COST_FUNCTION : String = "cost-function"
+ELEMENT_TASK : String = "task"
+ATTRIBUTE_PACKAGE : String = "package"
+ATTRIBUTE_TYPE : String = "type"
-DEBUG : boolean = false
+loadCostFunctionFile(filename : String) : void
+loadCostFunctionFile(file : File) : void
-setupCostFunctions(costFunctionList : NodeList) : void
-setupTaskMapping(costFunctionClass : Class<? extends CostFunction>, taskList : NodeList) : void

CostFunctionClassUtils
~getCostFunctionClass(className : String) : Class<? extends CostFunction>
~getCostFunctionConstructor(costFunctionClass : Class<? extends CostFunction>) : Constructor<? extends CostFunction>
~newCostFunction(constructor : Constructor<? extends CostFunction>) : CostFunction
~getClass(className : String) : Class<?>
-instantiate(constructor : Constructor<T>, parameters : Object ...) : T

Illustration 33: Object Construction Application Cost Functions

3.3 Running the Application

The "main" class is `sim.collbing.main.Launcher`. The main program arguments are listed below.

Argument Type	Sample Value
Environment file	<code>configs/CollBingEnvironment.xml</code>
Agent file	<code>configs/CollBing-Agents.xml</code>
Cost Function file	<code>configs/CollBing-CostFunctions.xml</code>

Create a run configuration in Eclipse with these arguments. The environment window will appear as in 3.1 Object Construction Application Description Illustration 16 above. Change the "0" to a higher number if desired to slow down the simulation (500 in the figure), then click start. An organization window appears for each agent and the scenario begins to execute.

3.4 Application Test Results

This section describes the testing performed of individual agent plan and capabilities, the proposed experimental design, and leaves a section for experimental results to be filled in later. The results are not available because time ran out to provide the `demircf.MessageOutput` implementation that would support look up of OMACS agent, goal, and role objects. Also code must be written to relay the `demircf.Message` objects to `DEMiRCF`, using the `AchievedMsg` to update `GMoDS`.

3.4.1 Testing Plans and Capabilities

As stated in 3.2.3 Agents above, the `BasicCBAgent.robotCode` method is structured to act as a test harness for `CBExecutionPlans` if `BasicCBAgent.TEST_PLANS` and `AbstractPlanCode.TEST` are set to `true`. Each concrete `AbstractPlanCode` class' `doInitialization` method has a `TEST` behavior that hard-wires the relevant parameters to data consistent with the project Environment file. Select the agent to test by editing the `CollBing-1Agent.xml` file. Select the plan to test by editing the agent class `getTestPlan` method to return the desired plan.

I tested all plans using the above methodology, in scenarios containing 1 agent.

3.4.2 Experimental Design

We plan to use an experimental design that closely follows Sariel and Balch [2] (pp. 6-7). They used two types of experiments:

1. Vary rates of message loss.
2. Vary agent failures.

It may also be wise to randomize the placement of the agents and the objects in the environment.

3.4.3 Experimental Results

TBD

4 References

1. Sariel, S., "An Integrated Planning, Scheduling and Execution Framework for Multi-Robot Cooperation and Coordination," PhD Thesis, Istanbul Technical University, Turkey, 2007.
2. Sariel, S. and Balch, T., Robust Multi-Robot Coordination in Noisy and Dangerous Environments, tech. report GIT-GVU-05-17, GVU Center, Georgia Institute of Technology, 2005.
3. Sariel, S., Balch, T., and Erdogan, N., "Incremental Multi-Robot Task Selection for Resource Constrained and Interrelated Tasks," *IEEE/RSJ Intl. Conf. On Intelligent Robots and Systems (IROS)*, 2007.
4. S. DeLoach, "Modeling Organizational Goals using the Goal Model for Dynamic Systems (GMoDS)," Lecture, Kansas State University, 2008.
5. S. DeLoach, "Organization Model for Adaptive Computational Systems," Lecture, Kansas State University, 2008.