MICE Users Guide

Thomas A. Montgomery, Jaeho Lee, David J. Musliner, Edmund H. Durfee
Daniel Damouth, Young-pa So, and the rest of the UM-DIAG
University of Michigan Distributed Intelligent Agents Group (UM-DIAG)
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109

January 31, 1992

Abstract

The Michigan Intelligent Coordination Experiment (MICE) testbed is a tool for experimenting with coordination between intelligent systems under a variety of conditions. In this document, we describe how to use the MICE system. We begin with a discussion of the design decisions that determine what can and cannot be done in MICE. Then we present a procedure for running a MICE experiment including the design of a simulation environment, the implementation of the intelligent agents for the environment, and the execution of MICE. The user interface that helps gather the results of experiments is described next. We conclude with examples from a predator/prey implementation.

[0]This research was sponsored, in part, by the National Science Foundation under grant IRI-9010645, by the University of Michigan under a Rackham Faculty Research Grant, and by a Bell Northern Research Postgraduate Award.

Contents

1 Design Decisions

The Michigan Intelligent Coordination Experiment (MICE) testbed simulates a two-dimensional world in which intelligent agents can interact. MICE was designed to allow reproducibility in experimentation and to be flexible and computationally efficient. In addition, MICE has been designed to be as easy to use as possible.

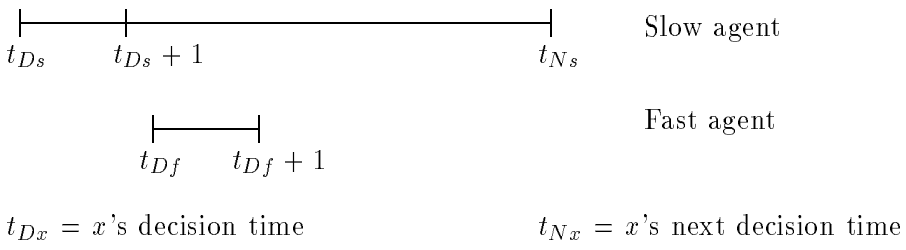
In developing MICE, we were primarily interested in building a testbed where we could simulate agents that are acting concurrently. Of major importance was the ability to examine the individual decisions that agents make and the context in which those decisions are made. Thus, rather than implementing agent concurrency at the operating system level (and thus relinquishing control over agent scheduling), we designed MICE as a discrete event simulation, where events and actions in the environment take some amount of simulated time and each agent has a simulated clock.

Because MICE is responsible for modeling the environment in which agents act, it must ensure that the environment is *legal* (all environmental constraints are satisfied) at each simulated time. Whenever agents take actions that lead to an illegal situation (such as when 2 agents that cannot share a location move into the same location), MICE must resolve the situation using information about the agents and user-supplied predicates. For example, if 2 agents that cannot share a location attempt to move into the same location, a user-supplied predicate might cause MICE to resolve the conflict by returning them both to their previous locations. Assuming that the previous situation was legal (and that the initial situation specified by the user is always legal), using a resolution predicate that returns conflicting agents to their prior locations can always resolve a new situation into a legal situation, in the worst case returning every agent to its previous state.

MICE not only represents time in discrete units, but in the current implementation it also represents agent actions and environment locations and events as discrete entities. For example, when an agent moves to its adjacent northerly location, it is in its original location at one discrete time, and is in the adjacent location in the next discrete time. There is no concept of being "partway" between discrete locations. Thus, in our design we had to decide how a movement that takes more than one time unit should be simulated: When exactly does an agent make the transition? To simplify resolution, our current implementation simulates these transitions by having the agent move to the new location immediately, and then "resting" there for the remaining duration of the move. This means that an agent that decides at time t_i to move to an adjacent location will arrive at that location at time $t_i + 1$. If the move requires n time units, it must remain in that location until time $t_i + n$. An upshot of this decision is that a slowly moving agent can "claim" a location ahead of a quickly moving agent that decides to move into that location later, as shown in Figure 1.[1]

In summary, our emphasis in developing the current version of MICE was to provide a platform for simulating multi-agent environments where the simulations are reproducible, efficient, and where records of discrete agent actions and interactions can be saved for later inspection. MICE accomplishes these goals through discrete event simulation. Because MICE also allows agents to specify some amount of simulated time spent reasoning, MICE provides a platform for studying issues in real-time decisionmaking. Real time constraints can be imposed by MICE using the `*real-time-knob*` (see Section 6.4), or the agents can impose such constraints on themselves. For example, the function that decides what an agent should do can record the times it begins and ends computation, and then map the elapsed real-time spent into some number of simulated time units (where different ratios of real to simulated time will change the severity of how quickly agents must reason). Dynamic environments can also be simulated by implementing inanimate objects as

[1] On the other hand, simulating an agent to arrive at its new location at the last second would be similarly problematic: It would "hold on" to its old location well after we would have thought it would have left it.



A slowly moving agent decides at time t_{Ds} to move to some location. It requires $t_{Ns} - t_{Ds}$ time units to complete the move, but is simulated to arrive one time unit after it makes the decision ($t_{Ds}+1$). It then cannot move from there until t_{Ns}. A quickly moving agent decides at t_{Df}, which is after t_{Ds}, to move to the same location. It would arrive there one time unit later at $t_{Df}+1$. However, because the slowly moving agent is already occupying that location, MICE resolves the situation by disallowing the quickly moving agent's move.

Figure 1: Timing of Actions of Differing Durations.

agents. For example, in a blocks world we can implement each block as an agent, and blocks might act so that they sometimes move unexpectedly or slip from some robot-agent's grasp.

2 MICE and Agents

To use MICE properly, it is imperative that the user realize the clear delineation MICE makes between what goes on outside an agent versus what goes on inside of it. As a simulator for multi-agent worlds, MICE is only concerned with what goes on outside of an agent. MICE represents an agent using the features described later in this manual. All of these features are concerned with what actions the agents can take, how long they take, what happens when incompatible actions are taken, how agents should look, etc. MICE does *not* represent, or even care about, what goes on inside of an agent, in terms of what the agent knows, what it remembers of what it (and others) have done before, how it uses sensory information, how it makes decisions, etc. In other words, the reasoning that an agent does is cleanly separated from how an agent affects (and is affected by) the simulated physical world.

Therefore, in implementing the decision making part of agents for MICE, the user will generally define knowledge representations and functions that are separate from those of MICE. As illustrated in the manual, and in example code supplied with MICE, what typically happens when MICE invokes an agent (by calling a function specified by the user which gets passed the agent representation that MICE uses), is that the invocation function retrieves the representation used by the agent's cognitive component.

One more word about agents and terminology in this manual. Agents are referred to in numerous ways. As mentioned above, each agent has its own data structure that MICE uses when manipulating the physical agent. Each agent also has its own unique name (generally supplied by the user). Finally, each agent has a type, which might be unique or might be shared with other agents. By defining types (classes) of agents, we can specify interactions among agents more compactly. In this manual, we attempt to make it clear what is expected when an "agent" is referred to, manipulated, or passed as an argument.

3 Building an Environment

Building an environment in the MICE system requires the specification of the world itself (the grid features and communication channels), the characteristics of agents within the world, and the interactions between agents. This information is specified in an environment file (Section 5.1).

Grid Description. The first step in building an environment is deciding on the features of the world that the agents will occupy. This includes the size of the world (its dimensions) and the features of locations within the world. For example, in simulating a fire fighting scenario, it may be decided to specify locations according to their content such as trees, water, and roads. In another simulation, it may be decided that the content does not matter, but that elevation is relevant. In any case, the important issue in deciding on grid features is not to try to mimic the real world exactly, but rather to find the features of the real world that have an impact on the coordination issues faced by the intelligent agents. Details on how to implement a grid are presented in Section 3.1.

Communication Channels. Communication between agents is often desired when experimenting with coordination techniques. MICE provides communication facilities by allowing the user to define the different communication media and their various characteristics. Agents communicate with each other by issuing **:SEND** and **:RECV** commands. Communication in MICE is channel-based, and the characteristics and participants of communication are defined as part of the channel. Further details are in Section 3.2.

Agent Types. Next, the different types of agents must be determined. This includes deciding on a classification of the agents and determining the characteristics shared and not shared by agents of the same type. Such characteristics determine the abilities of the agent in the environment and may relate back to the grid description (for example the speed of an agent may depend on the terrain being covered).

Agents are defined through calls to `create-agent` which accepts a number of optional keyword arguments. Those that are the most general and are used in almost all applications include `:NAME`, `:LOCATION`, `:ORIENTATION`, `:TYPE`, `:SENSORS`, `:MOVE-DATA`, `:LINK-COST-ALIST`, `:UNLINK-COST-ALIST`, and `:DRAW-FUNCTION`. Other agent characteristics that are used in a large number of domains, though all of them might not be used in any one application, include `:BLOCKED-BY-TYPES`, `:DOMAIN-VARIABLES`, `:AUTHORITY`, `:CREATE-P`, `:CREATE-FUNCTION`, `:REMOVE-P`, `:REMOVE-FUNCTION`, `:ACTIVATE-P`, `:ACTIVATE-FUNCTION`, `:INACTIVATE-P`, and `:INACTIVATE-FUNCTION`. Finally, `:CAPTURE-TYPES` and `:CAPTURED-BY-TYPES` are domain specific characteristics that are included due to the system's early emphasis on predator-prey scenarios. Since the same results can be obtained using the more general `:DOMAIN-VARIABLES`, these options may be removed in a future release. Further details on the parameters to `create-agent` are presented in Sections 3.3 and 3.4.

Agent Type Interactions. In addition to the relationship between the agents and the environment, the relationship between agent types must be specified. This includes determining what happens when two or more agents attempt to move to the same location, how the agents involved are affected when they collide, how other spatial relationships affect agents or the environment, how the presence of an agent might obstruct the scanning of another agent, and what types of agents can capture what other types. Implementation details for agent interactions can be found in Section 3.4.

3.1 Defining the Grid

The MICE world is a two-dimensional grid, and each location has a corresponding **grid-element**. The size of the grid can be modified as described in Section 5.1. Unless explicitly modified, each grid-element assumes that locations have no special features. The function `get-grid-element` retrieves the grid-element for some specified location. The fields of a grid-element are settable and selectable using the following accessor functions: `grid-element$features`, an association list of user-defined features and their values; `grid-element$agents`, a list of the agents currently occupying the location; and `grid-element$draw-function`, used by the graphics routines in deciding how to represent the location graphically.

For example, the following code appears in the environment file and modifies the grid to represent a north-south wall in the middle of the grid. This wall blocks out predators, but prey can move through it

```
(setf (grid-element$features (get-grid-element (make-location :x 10 :y 8) t))
 (setf (grid-element$features (get-grid-element (make-location :x 10 :y 9) t))
  (setf (grid-element$features (get-grid-element (make-location :x 10 :y 10) t))
   (setf (grid-element$features (get-grid-element (make-location :x 10 :y 11) t))
    (setf (grid-element$features (get-grid-element (make-location :x 10 :y 12) t))
          (acons :BLOCKED-TYPES (list :PREDATOR) nil)))))))
```

3.2 Defining Communication Channels

Communication channels are implemented by placing calls to **create-channel** in the environment file. **create-channel** accepts the following keyword parameters to specify channel characteristics.

:NAME *channel-name*
The *channel-name* is a unique identifier for each channel that should be made up of a combination of capital letters and numerals. It can be specified as a string or a symbol, although MICE converts it to a symbol for internal use. If no name is specified, MICE creates a name of the form C< *integer* >.

:AGENTS *agent-name-list*
The *agent-name-list* is a list of agent names participating in the channel. Each name should be for an agent that has already been created. Communication through the channel is possible only among participating agents.

:DELAY *communication-delay*
The *communication-delay* is a positive integer characterizing the communication time delay of the channel. The default value is 1.

:CAPACITY *communication-capacity*
The *communication-capacity* is a positive integer representing the communication capacity of the channel in terms of number of messages per unit time. **nil** represents *infinite* capacity. The default value is **nil**. If the agents using the channel together try to send more messages than the communication capacity, only the highest priority messages are sent. The other messages are handled according to the value of `:FAILURE-MESSAGE-PRIORITY` of the channel; the :STATUS of failed messages are set to `:OVER-CAPACITY`.

:RELIABILITY *probability-of-successful-transmission*
The *probability-of-successful-transmission* is a real value between 0 and 1. Reliability of the channel is summarized as the probability that a message is transmitted successfully over the channel. The default value is 1. The failed messages are handled according to the value of `:FAILURE-MESSAGE-PRIORITY` of the channel; the `:STATUS` of the failed messages are set to `:FAILURE`.

:RANGE *communication-region*
The *communication-region* is a region created by the **make-region** function that includes minimum and maximum x and y displacement from the agent's current location. If the range is not symmetric around the agent, then the `:ORIENTATION-SENSITIVE-P` slot should contain `t`, in which case the range specified is assumed to be for the agent when facing north. MICE will compute the appropriate region for different orientations of the agent. The default value of `:RANGE` is **nil** specifying *infinite* communication range. If the *hearer* is out of the communication range of the *speaker*, transmission of the message fails with the message `:STATUS` set to `:OUT-OF-RANGE`.

:OBSTRUCTED-BY *obstruction-function*
The *obstruction-function* is a function that takes a grid-element as an argument and returns a number between 0 and 1 (indicating the width of the obstruction in the grid location), or **nil** if there is no obstruction in the location. If the communication between *speaker* and *hearer* is obstructed, transmission of the message fails with the message `:STATUS` set to `:OBSTRUCTED`.

:FAILURE-MESSAGE-PRIORITY *priority-for-failed-messages*
The *priority-for-failed-messages* is a priority value for the failed messages. Transmission of messages fails because of limited capacity (`:CAPACITY`), unreliability (`:RELIABILITY`), limited communication range (`:RANGE`), and communication obstruction (`:OBSTRUCTED-BY`). Since received messages are ordered on the priority of the messages, setting the `:FAILURE-MESSAGE-PRIORITY` value high makes the failed messages come first in the receive queue of the agent when the messages are sent back to the *speaker* because of communication failure. If the value is negative, the failed messages are dropped and are not sent back to the *speaker*.

:TIME-TO-SEND *time*
The *time* is a positive integer value specifying the time required for an agent to **:SEND** a message (place it) on the channel. The default value is 0.

:TIME-TO-RECEIVE *time*
The *time* is a positive integer value specifying the time required for an agent to **:RECV** a message (take it off the channel). The default value is 0.

A simple example of a channel is:

```
(create-channel :NAME 'channel-1
                :AGENTS '(PRED1 PRED2 PRED3 PRED4)
                :DELAY 1
                :CAPACITY nil
                :RELIABILITY 1.0
                :RANGE NIL
                :ORIENTATION-SENSITIVE-P :UNKNOWN
                :OBSTRUCTED-BY nil)
```

Here, a channel named 'channel-1' is established for communication between four agents named PRED1 through PRED4. Messages take one time unit to propagate to their destination, and the channel has infinite capacity and never loses messages. Its range is infinite in all directions, so orientation of the agent will not affect which agents can receive messages from it. Communication cannot be obstructed.

3.3 Defining Agents

Agents are implemented by placing calls to `create-agent` in the environment file. Create-agent accepts keyword parameters to specify agent characteristics. Most of the parameters are described in this section, although some are deferred to Section 3.4. Those parameters that should almost always be specified include:

:NAME *agent-name*
The *agent-name* is a unique identifier for each agent that should be made up of a combination of capital letters and numerals. It can be specified as a string or a symbol, although MICE converts it to a symbol for internal use. If no name is specified, MICE creates a name of the form A<*integer*>.

:LOCATION *starting-location*
The *starting-location* is a MICE-defined structure created with `make-location` that contains two slots: `:X-LOC` and `:Y-LOC`. Starting locations can be randomly generated within a region of the grid by using `make-random-location`.

:ORIENTATION *orientation*
The *orientation* is a keyword that indicates the direction that the agent is facing (`:NORTH`, `:SOUTH`, `:EAST` or `:WEST`). If no orientation is specified, then it is assumed that the agent is symmetrical and all orientation-related predicates are ignored.

:TYPE *agent-type*
The *agent-type* is a user defined keyword specifying the class of agents that this agent belongs to.

:SENSORS *sensor-list*
The *sensor-list* is a list of sensors that are available to the agent. Sensors are created using `make-sensor-data` which takes keyword arguments to specify their `:RANGE`, the amount of time consumed in using them (`:TIME`), whether they are `:ORIENTATION-SENSITIVE-P`, the type of information that they pick up (`:INTERESTING-P`), and information about objects that they cannot see past (`:OBSTRUCTED-BY`). The `:RANGE` should be a region created by `make-region` that includes minimum and maximum x and y displacements from the agent's current location (`:X-MIN, :Y-MIN, :X-MAX`, and `:Y-MAX`). If the range is not symmetric around the agent, then the `:ORIENTATION-SENSITIVE-P` slot should contain **t**, in which case the range specified is assumed to be for the agent when facing north. MICE will compute the appropriate region for different orientations of the agent. The `:TIME` is an expression that, when evaluated, should return an integer. The `:INTERESTING-P` slot contains a function that takes a grid-element as an argument and returns either **t** or **nil** depending on whether the sensor is allowed to detect the contents of that grid-element based on those contents. The `:OBSTRUCTED-BY` field is a function that takes a grid-element as an argument and returns a number between 0 and 1 (indicating the width of the obstruction in the grid location), or **nil** if there is no obstruction in the location. If `:SENSORS` is not specified, then the agent is given a default

sensor which has a time cost of zero, is not obstructed, and has a range of -5 to 5 in both the x and y directions.

:CHANNELS *channel-list*
The *channel-list* is a list of channels that are available to the agent. Channels are created using **create-channel** which is explained in Section 3.2.

:MOVE-DATA *move-data*
The *move-data* is a structure created by `make-move-data` that indicates the amount of time it takes to move in each direction (`:NORTH, :SOUTH, :EAST, :WEST`, or `:FORWARD, :BACKWARD, :LEFT, :RIGHT`), and the time it takes to `:ROTATE-ONE-QUADRANT`. The values should evaluate to either an integer corresponding to the time needed, or **nil** if movement in that direction is forbidden. The default for each is 1.

:LINK-COST-ALIST *association-list*
The *association-list* pairs agent types with a simulated time cost that the agent incurs when linking with agents of that type. The simulated time cost is an expression that, when evaluated, returns an integer. In future releases, this cost will be allowed to vary depending on the type of link being formed.

:UNLINK-COST-ALIST *association-list*
The *association-list* pairs agent types with a simulated time cost that the agent incurs when unlinking from agents of that type. The simulated time cost is an expression that, when evaluated, returns an integer. In future releases, this cost will be allowed to vary depending on the type of link being removed.

:DRAW-FUNCTION *icon-function* [KEY argument]*
The *icon-function* is a function which draws the desired icon using device-independent graphics calls defined in di-graphics.lisp. A set of common icons is available from icons.lisp, including: **rectangle-icon, circle-icon, triangle-icon, clock-icon, square-clock-icon, hero-icon,** and others. Most of these icon functions can take one or more *keyword argument* pairs to specify their appearance. For example, most icons can be drawn as solid figures instead of hollow by specifying `:FILLED t`. The hour to display on the clock icons is specified by `:HOUR 1-12`. Some of the icons also take a `:LABEL "string"` argument, which will display the label in the center of the icon. Figure 2 shows how some of the available icons look, as used in the testicons.env file.

When the icon function is actually called by MICE, its first argument is the grid location in which it should draw the icon. If the icon is representing an agent, the agent structure is passed in under the `:AGENT` keyword, so that the icon function can examine the state of the agent and alter its graphical representation as necessary. For example, the `hero-icon` function draws the arm of the Hero pointed towards the direction the Hero is headed, and changes the length of the arm if the Hero is linked. See icons.lisp for clues on writing new icon functions. Users should feel free to create new icons: they will not require any changes to MICE itself.

Other agent characteristics are important for a large number of domains, but may be left unspecified in some cases:

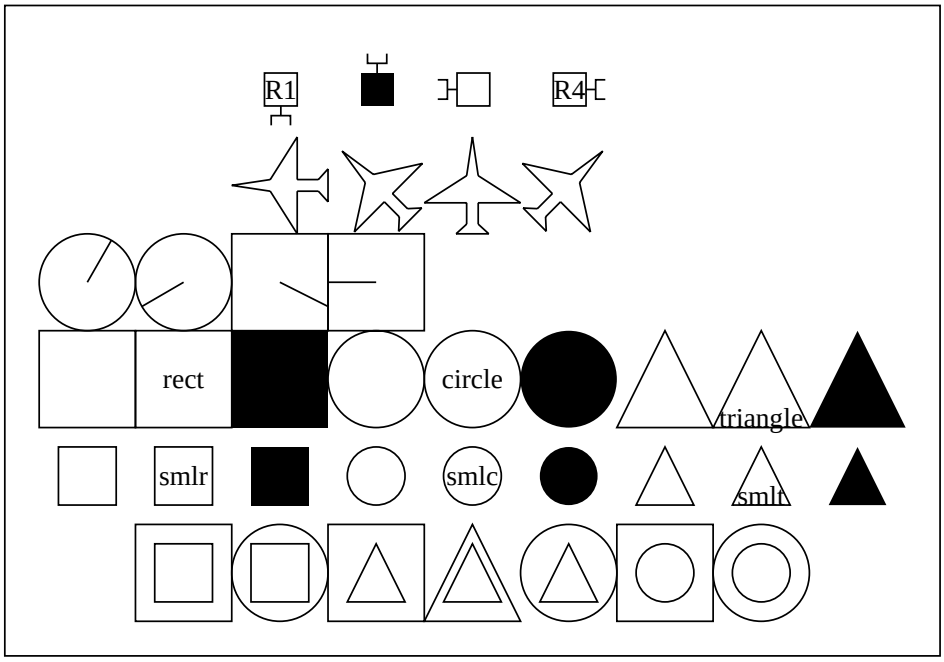


Figure 2: PostScript output of example icons found in testicons.env.

:BLOCKED-BY-TYPES *agent-type-list*
`:BLOCKED-BY-TYPES` is a list of those agent types that this agent cannot share a location with. It is used to trigger calls to the user-specified collision function (see Section 3.4).

:AUTHORITY *agent-authority*
The *agent-authority* is an integer that represents the authority of an agent relative to the other agents. If not specified, an agent's authority defaults to the integral portion of its name or to 1 if its name does not contain numerals (for example, RED5 would have a default authority of 5, while AGENTA would default to 1).

:DOMAIN-VARIABLES *assoc-list*
This field contains any domain-specific information desired in an association list. For example, an implementation that requires the use of strength and size information for agents would fill this slot with something like: ((`:STRENGTH` . 5) (`:SIZE` . 5)).

Finally, there are some agent characteristics that are currently supported by MICE for historical reasons, and are of particular use in simulating predator/prey environments.

:CAPTURE-TYPES *agent-type-list*
A list of the agent types that this agent can help capture.

:CAPTURED-BY-TYPES *agent-type-list*
A list of the agent types that can capture this agent.

3.4 Defining Agent Interactions

Agents can interact in various ways. Sometimes, the actions of individual agents can lead to combinations of actions that violate environmental constraints. When constraints are violated, the situation must be resolved; for example, if two agents that cannot occupy the same location simultaneously move to the same place, MICE must resolve the situation. At other times, the agents can enter situations that trigger some response in the environment. For example, if we want our simulation to remove agents once they are captured (surrounded by agents of another type), we must supply MICE with predicates to test for relevant situations and functions that specify the consequences of the situation. Information about agents' interactions are specified in the following fields of the agents, where each field contains some function or predicate chosen from the growing library of domain-specific possibilities or created by the user.

:COLLISION-FUNCTION *function-definition-assoc-list*
The *function-definition-assoc-list* pairs each agent type that the current agent might collide with, with the function to resolve the collision. An agent-type specifier `:ALL` indicates that the associated function applies to all collisions. Each such function takes as arguments the structures of the 2 agents that have collided (either by moving to the same location or by switching locations which implies that they must have passed through one another) and the time of the collision. Unless the agents' `:BLOCKED-BY-TYPES` fields name each other's types, the agents cannot collide. If they do block each other, then the **default-collision-function** will move them back to where they were at the previous time. Essentially, they bounce off one another to return to where they were. Another option is to use the agents' `:AUTHORITY` field to decide which agent should be given preference. The **authority-collision-function** gives the preferred agent its desired location, while the other agent is returned to its previous location (unless it had not moved, in which case it is "pushed" in front of the preferred agent). Users are also free to define their own collision functions.

Once any constraint violations have been resolved, MICE checks other environmental interactions that might be precipitated in the new situation.

:OVERLAP-PREDICATES *function-definition-assoc-list*
The *function-definition-assoc-list* pairs each agent type that the current agent might overlap (currently, share a location) with, with a function to apply in that instance. An agent-type specifier `:ALL` indicates that the associated function applies to all collisions. Each such function takes as arguments the structures of the 2 agents that are overlapping and the time of the overlap, and determines the side-effects of the overlap. For example, the sharing of a location between 2 agents might cause the authority of one to rise and the other to fall (as if they were exchanging power between them).

:CREATE-P *predicate*
The *predicate* takes as arguments the agent structure and the simulated time, and returns non-nil if the agent should create some agent(s), causing the `:CREATE-FUNCTION` to be invoked. For example, if an agent survives for a certain amount of time, it might generate a clone of itself as a form of reproduction.

:CREATE-FUNCTION *function*
The *function* takes as arguments the agent structure and the simulated time. The function should

include one or more calls to `create-agent`, and should return a list of the results of those calls (containing the names of the created agents).

:REMOVE-P *predicate*
The *predicate* takes as arguments the agent structure and the simulated time, and returns non-nil if the agent should remove some agent(s), possibly including itself. When non-nil, it causes the `:REMOVE-FUNCTION` to be invoked. For example, if an agent is surrounded by predators, it might remove itself because it has been captured.

:REMOVE-FUNCTION *function*
The *function* takes as arguments the agent structure and the simulated time, and returns either a list of agent structures or a single agent structure of the agent(s) that should be removed from the environment. For example, the **mice-self** function returns current agent's data structure.

:ACTIVATE-P *predicate*
The *predicate* takes as arguments the agent structure and the simulated time, and returns non-nil if the agent's status should be changed to `:ACTIVATED`. When non-nil, it causes the `:ACTIVATE-FUNCTION` to be invoked. For example, if an agent has "rested" for a while, it might re-activate itself.

:ACTIVATE-FUNCTION *function*
The *function* takes as arguments the agent structure and the simulated time. It performs any side-effects that are desired when an agent is being activated.

:INACTIVATE-P *predicate*
The *predicate* takes as arguments the agent structure and the simulated time, and returns non-nil if the agent's status should be changed to `:INACTIVATED`. When non-nil, it causes the `:INACTIVATE-FUNCTION` to be invoked. For example, if an agent has used up all of some regenerable resource, it might become inactive until that resource is regenerated. Note that MICE does not automatically restrict the actions of an inactivated agent, it is up to the user to decide what it means for an agent to be inactive.

:INACTIVATE-FUNCTION *function*
The *function* takes as arguments the agent structure and the simulated time. It performs any side-effects that are desired when an agent is being inactivated.

3.5 An Example Agent Specification

The specification below, taken from an environment file, creates an agent named PRED1 that is of type `:PREDATOR`. It can capture agents of type `:PREY`, and also is blocked by (cannot share a location with) agents of type `:PREY`. If it and an agent that can block it attempt to move into or through each other, their collision is resolved by a function (supplied with MICE) called *authority-collision-function* that lets the agent with higher authority get its way. The authority of this agent is specified as 10. It is created at simulated time 0 at a randomly chosen location within the 20 by 20 grid. It can communicate on either or both of channel-1 and channel-2, and has a sensor whose range extends 10 units in each of the four directions. When MICE determines that this agent should provide more commands, MICE calls the function named *manager-contractor-agent*.

When this agent is drawn on the screen, it is drawn like a circular clock face with a hand pointing to one o'clock.

```
(create-agent :NAME "PRED1"
              :TYPE :PREDATOR
              :CAPTURED-BY-TYPES ()
              :CAPTURE-TYPES '(:PREY)
              :BLOCKED-BY-TYPES (list :PREY)
              :COLLISION-FUNCTION
              (acons :ALL '#'authority-collision-function nil)
              :AUTHORITY 10
              :CREATION-TIME 0
              :LOCATION (make-random-location :X 20 :Y 20)
              :CHANNELS '(channel-1 channel-2)
              :SENSORS
              (list (make-sensor-data
                     :RANGE (make-region :X-MIN -10 :Y-MIN -10
                                         :X-MAX 10 :Y-MAX 10)))
              :INVOCATION-FUNCTION 'manager-contractor-agent
              :DRAW-FUNCTION '(clock-icon :hour 1))
```

4 Agent Implementation and Interface

MICE has been designed to separate agent implementations from the simulation of the environment in which agents live. Because of the desire to allow agents with very different reasoning architectures to reside in MICE (and with each other), we have enforced a very clearly delineated interface between agents and MICE.

4.1 Agent Invocation

As MICE simulates the concurrent activities of agents, it must interleave the activities of the agents based on each agent's simulated clock time (see Section 5.3). When an agent is chosen for execution, MICE executes the agent by calling the agent's `:INVOCATION-FUNCTION`. The `:INVOCATION-FUNCTION` is specified by the user when defining an agent as a parameter of the **create-agent** function.

An agent's invocation function takes as an argument the current agent's data-structure. This allows different agents to use the same invocation function since the function receives information about which agent is actually being executed. In future CLOS-based releases, this argument to the invocation function will change to provide better information hiding. The invocation function should return a list of commands to MICE indicating the decisions that the agent has made about its actions at this simulated time.

4.2 Agent Commands to MICE

Currently, agents can issue commands to MICE that represent decisions that the agents have made regarding motion, perception, interactions, and time. These commands can include requests to **:MOVE** to an adjacent location, to **:ROTATE** in order to change orientation, to **:SCAN** some region, to **:LINK** to another agent, to **:UNLINK** from another agent, to take no action at the

current time (the **:QUIESCENT** command), to take no action for a specified length of time (the **:NULL-ACTION** command), to spend a certain amount of time **:REASONING**, or to **:STOP**.

MICE allows two sources of simulated-time costs. One source is the cost associated with a particular action, such as how long it takes to move, scan, or link. The information used to compute these costs are specified when an agent is implemented. The other source is the cost associated with deciding on that action. In some simulations, this source is considered negligible, and so these costs are always 0. However, in other simulated environments, the time needed to decide on an action adds to the overall time for taking the action, and might delay the initiation of the action. MICE allows the latter type of time cost via the **:REASONING** command, whereby an agent can specify that it has spent a certain amount of time reasoning. Alternatively, time spent reasoning can be charged by MICE by using the `*real-time-knob*` described in Section 6.4.

For example, let us say an agent spent 2 simulated time units deciding to move `:NORTH`. Its invocation function would return 2 commands, one indicating **:REASONING** for 2 time units, and then a second requesting a **:MOVE** `:NORTH`. These are buffered by MICE and executed at the proper time. Of course, if the agent is behaving in a very dynamic environment, it might want to double-check between the **:REASONING** and **:MOVE** commands to make sure that the **:MOVE** is still valid. This capability must be supported by the agent itself: Once a sequence of commands are sent to MICE, they are executed entirely before the agent's invocation function is once again called. However, the user can implement an agent such that it internally buffers a sequence of commands (possibly with associated validity conditions) and issues these one at a time to MICE. This way, the agent is given a chance to double-check the situation before issuing the next command.

A command to MICE is essentially a request for some action to take place. These actions are tentative because concurrent actions by several agents might conflict. For example, two agents that cannot share locations can issue commands that would result in their being in the same location. In this case, the agents are tentatively moved and their simulated clocks updated, but MICE later recognizes the constraint violation and resolves it in the user-specified way (by default, it moves them back to their original positions). The time costs of the moves, however, are not undone: The agents have wasted some amount of time by taking conflicting actions.

The commands to MICE and their related arguments are:

:MOVE *direction*
The *direction* is currently one of `:NORTH`, `:SOUTH`, `:EAST`, `:WEST`, `:FORWARD`, `:BACKWARD`, `:LEFT`, `:RIGHT`, or **nil** (stay in the same location). This causes the agent to be tentatively moved to the adjacent location in the direction indicated (in the case of `:FORWARD`, `:BACKWARD`, `:LEFT`, and `:RIGHT`, the direction will depend on the agent's current orientation). The time costs of the move are computed from the `:MOVE-DATA` information provided by the user when defining the agent.

:ROTATE *direction number-of-quadrants*
The *direction* is generally one of `:RIGHT` or `:LEFT`, although the agent can request an absolute direction (`:NORTH`, `:SOUTH`, `:EAST`, `:WEST`) which will cause the agent to turn to that direction (arbitrarily to the left if it must turn all the way around). The *number-of-quadrants* indicates how far to turn (1 quadrant equals 90 degrees). This causes the agent's orientation to be tentatively changed. The time costs of the rotation are computed from the `:ROTATE-ONE-QUADRANT` slot of the agent's `:MOVE-DATA` information provided by the user when defining the agent.

:SCAN *sensor*

The *sensor* is a sensor data structure from the list of the agent's sensors (see Section 3.3) or the keyword `:ALL`. The time costs are computed from the sensor data structure. MICE simulates the execution of the provided sensor by searching through the appropriate grid-elements and storing the scanned information as a list of grid-descriptions. The agent can subsequently retrieve this information using the function `read-and-reset-scanned-data` (Section 4.3).

:AFFECT *location* ([:FEATURES *nil*] [:DRAW-FUNCTION *nil*])
The *location* is either a location in the grid or a relative direction (see **:MOVE**). If the keyword `:FEATURES` is specified, the features field of the grid element specified by *location* will be changed to the given argument. Likewise, if the keyword `:DRAW-FUNCTION` is specified, the grid element's draw function will be changed to the given argument. This command takes no simulated time.

:LINK *linked-to-agent link-type*
The *linked-to-agent* is the name of an agent to which the current agent is attempting to link. The *link-type* is currently one of `:FRONT`, `:LEFT`, `:RIGHT`, `:BACK`, `:SHARED-LOC`, `:NORTH`, `:SOUTH`, `:EAST`, `:WEST`, or `:NEXT-TO`, indicating the spatial relationship of the linked-to-agent relative to the current agent at the next time unit. Links to front, back, left, and right are orientation dependent, so turning will "swing" a linked agent around. `:SHARED-LOC` allows an agent to link to another that should occupy the same location. Links to north, south, east, and west keep the linked agent in that direction from the linking agent regardless of the linking agent's orientation. `:NEXT-TO` allows linking to an adjacent agent in any direction (north, south, east, west), and from that point on maintains the link in only that specific direction. The time costs of linking are computed from the user-supplied information in the agent's `:LINK-COST-ALIST` slot.

:UNLINK *linked-to-agent*
The *linked-to-agent* is the name of an agent from which the current agent is attempting to unlink. The time costs of unlinking are computed from the user-supplied information in the agent's `:UNLINK-COST-ALIST` slot.

:REASONING *time*
The *time* is the simulated time spent reasoning. **:REASONING** only increments the agent's simulated time by the amount specified, and has no other effects.

:NULL-ACTION *time*
Increments the agent's clock by the amount specified by *time*. Has no other effects.

:QUIESCENT
Same as **:NULL-ACTION**, except that the amount of idle time is exactly 1 time unit.

:STOP
Permanently removes the agent from the set of simulated agents.

:SEND *channel-name type content* ([:PRIORITY *priority*] [:HEARER *agent-name*])
The agent sends a message through the channel *channel-name* with the *priority*. The message is of *type* with *content*. User can specify any *type* and *content* of the message as arguments to this command. If the *hearer* is not specified explicitly as a keyword argument, the default is for all participants of the channel including the speaker to receive the message.

:RECV *channel-name* ([:COUNT *number*] [:CLEAR *t-or-nil*])
The agent receives at most *count* messages from the *channel-name*. If the keyword argument *clear* is set to **t**, the remaining messages of the channel are cleared after returning at most *count* messages from the channel. If the number of messages to receive from the channel is greater than *count*, messages are selected based on the *priority* of the message. The default *count* value is infinity. The agent can subsequently retrieve received messages using the function **read-and-reset-received-messages** (Section 4.3).

One limitation of the current MICE commands is that a diagonal movement is a combination of discrete steps, leading to zig-zag movement. The distance an agent travels (and the time of its trip) between two diagonal locations is simply the manhattan distance. As a result, a round about route (n moves north then n moves east) takes as much time as moving "diagonally" between the locations.

Because some simulations might require intelligent route planning where the costs of moving diagonally should be less than the manhattan movement costs, we are adding that functionality. Currently available (but not thoroughly tested) are the additional commands:

:GOTO *x y break-p*
Plots a path from the agent's current location to location x, y and saves the sequence of individual move commands. Computes the overall movement time as the smallest integer greater than $\sqrt{time(\Delta x)^2 + time(\Delta y)^2}$. It then allows moves in the sequence to go "faster." It is recommended that **:GOTO** only be used when the times for movements in all directions are greater than 1, otherwise **:GOTO** might cause instantaneous moves. MICE cannot guarantee correct application of environmental constraints when moves are instantaneous. In addition, since MICE requires that time costs be specified as integers, greater resolution can be achieved by scaling up the time costs. For example, consider moving to an adjacent diagonal location. If moves in the x and y directions take 1 time unit, then the diagonal move time equals the manhattan distance because $\sqrt{2}$ is rounded up to 2. However, if we scale all time units by a factor of 10 (moves in the x and y directions take 10 time units), then the diagonal move $\sqrt{200}$ is rounded to 15, which is a savings compared to the manhattan distance of 20.

The *break-p* argument is a predicate that is evaluated before each of the stored move commands. If the evaluation returns non-nil, then the agent's invocation function is called and the agent has an opportunity to insert new commands ahead of the remaining move commands. This is used, for example, to scan for obstructions that might have moved into view since the **:GOTO** command was first issued. If the **:GOTO** command should be aborted, the agent should return the **:INTERRUPT** command.

:INTERRUPT
Whenever MICE encounters the **:INTERRUPT** command for an agent, it discards any pending commands for the agent.

4.3 Interface Functions to MICE

Agents should generally interface to MICE via the commands specified above. However, certain MICE functions might allow more powerful (though potentially less structured) uses of MICE. Other MICE functions might be useful utility functions for implementing agents. The functions available in the MICE external interface (extint.lisp) are described below.

create-agent &REST keyword-arguments
Creates a new agent and incorporates it into the simulation. The keyword arguments correspond to the fields of an agent (Section 3.3). Specifically, the new agent structure is added to two lists that MICE keeps. One list is called `*agent-schedule-queue*`, which is the list of agents that MICE cycles through during the discrete event simulation. Only agents on that list can be invoked. The second list is called `*all-agents*`, which holds structures for all agents created during a run of MICE. The distinction is that an agent might be removed due to interactions within MICE (such as a prey being captured and consumed by predators); the removed agent would be deleted from the `*agent-schedule-queue*`. However, to maintain history, and to allow runs to be saved and restored, that agent will remain on the `*all-agents*` list.

create-channel *&REST keyword-arguments*
Creates a new communication channel and incorporates it into the simulation. The keyword arguments are explained in (Section 3.2). The resulting structure gets added to the list `*mice-channels*`.

agent-structure *agent-name*
Returns the agent data-structure for the agent whose name matches the symbol *agent-name*.

read-and-reset-scan-data *&KEY (reset-value nil) (agent *current-agent*)*
Returns the list of grid-descriptions generated by the last **:SCAN** command to MICE. Each grid-description has information about its location (`grid-description$coordinates`), the agents at the location (`grid-description$agents`), and the features of the corresponding grid-element (`grid-description$features`). **read-and-reset-scan-data** also sets the buffer to *reset-value* after reading it. Note that using a different reset value can distinguish between having scanned since the last read-and-reset-scan-data but nothing was there (**nil**), versus not having scanned since the last read-and-reset-scan-data (reset-value). In particular, in cases where the same agent is invoked multiple times at the same discrete time (because, for example, scanning is simulated to take no time), such distinctions can be crucial to avoid infinite loops.

read-and-reset-received-messages *&KEY (reset-value :EMPTY) (agent *current-agent*)*
Returns the list of messages generated by the last **:RECV** command to MICE. Clears the buffer and resets the buffer with the *reset-value*. If no messages are generated by the last **:RECV** command, it returns **nil**. If the function is called subsequently at the same discrete time without another **:RECV** command, it returns current *reset-value*.

get-grid-element *location &OPTIONAL (create nil) &KEY x y*
Returns the grid-element for the specified location (or for location (x,y) if provided). If no grid-element exists for the location and the optional *create* argument is non-nil, then a new grid-element is created and returned.

find-agent-location *agent &OPTIONAL (time *current-time*)*
Returns the location of the agent at the given time. If the time is not supplied, returns the current location of the agent.

find-agent-orientation *agent &OPTIONAL (time *current-time*)*
Returns the orientation of the agent at the given time. If the time is not supplied, returns the current orientation of the agent.

find-agent-status *agent &OPTIONAL (time *current-time*)*
Returns the status of the agent at the given time. If the time is not supplied, returns the current status of the agent.

find-agent-linkages *agent &OPTIONAL (time *current-time*)*
Currently returns **t** if the agent is linked to at least one other agent at the given time (or at the current time if the *time* argument is not specified).

find-agent-other *agent &OPTIONAL (time *current-time*)*
Sometimes, the user wants to have recorded, for each simulated time, something else about each agent. To do this, the user should set the variable `*other-state-history-information-function*` to a function that returns whatever state information the user wants recorded for the agent at each simulated time. Then the function **find-agent-other** can be used to retrieve this information. **find-agent-other** currently returns **t** if the agent is linked to at least one other agent at the given time (or at the current time if the *time* argument is not specified).

move-agent *agent-structure direction*
Moves the agent in the desired direction. Returns the simulated time costs for that movement.

compute-new-location *old-location direction*
Given an old-location and a direction, generates a new location data-structure for the location adjacent to the old-location in the specified direction.

legal-location-p *location &KEY (agent nil)*
Checks to see whether the location is within the grid boundary. If the *agent* parameter is specified (an agent data-structure), it also checks to see if the agent is allowed in the location, based on the features of that location, by calling **agent-allowed-in-location-p**.

agent-allowed-in-location-p *location agent*
Checks to see if the agent (given as an agent data-structure) is allowed in the location, based on the features of the location.

legal-region-p *region &KEY (agent nil) x-min x-max y-min y-max*
Checks to see whether the region is within the grid boundary. Returns the allowable portion of the region, or **nil** if the entire region falls outside of the grid boundary. If the *agent* parameter is specified (an agent data-structure), it also checks to see if the agent is allowed in **every** location within the region, based on the features of the locations, by calling **agent-allowed-in-region-p** with the allowable portion of the region.

agent-allowed-in-region-p *region agent &KEY x-min x-max y-min y-max*
Checks to see if the agent (given as an agent data-structure) is allowed in **every** location within the region (based on the features of the locations).

location-in-region-p *location region*
Checks to see if the location is in the region.

regions-overlap-p *region1 region2*
Checks to see if any locations are common to region1 and region2.

find-movement-time *agent direction*
Determines the time costs for the agent (given as an agent data-structure) to move in the given direction.

rotate *agent direction number-of-quadrants*
Changes the orientation of the agent by rotating it the given number of times in the direction provided. Returns the time costs for the rotation (the number of quadrants times the agent's rotate cost/quadrant).

null-action *agent time*
Increments the agent's clock by the provided amount of time.

scan-mice-region *scan-regions sensor-location &KEY (interesting-p #'live-agent-at-location-p) obstructed-by x-min y-min x-max y-max*
Scan-regions is a list of regions to be investigated, *sensor-location* is the current location of the sensor performing the scanning (used to determine which parts of the region are obstructed), *interesting-p* is a predicate used to test the features of a location to decide whether the location is of interest, *obstructed-by* is an optional predicate used to test the features of a location to decide whether the sensor can penetrate through the location, and *x-min, y-min, x-max,* and *y-max* provide an alternative means for specifying the region for inspection. This function returns a list of grid-descriptions within the specified region that meet the interesting-p criteria and which are not obstructed by other objects in the region.

live-agent-at-location-p *grid-element*
Checks the grid-element given (corresponding to a particular location) and returns non-nil if any agents in that location currently have a status of `:ACTIVE`.

scan-for-agent-grid-descriptions *agent*
Given an agent doing the scanning, checks its list of sensors and its current location and calls scan-mice-region with that information (all other arguments to scan-mice-region receive their default values).

get-visible-grids *x-sensor y-sensor x-min x-max y-min y-max interesting-p obstructed-by*
Returns grid-descriptions for all interesting locations within the region defined by (x-min x-max y-min y-max) that are visible from the sensor location (x-sensor y-sensor). *(This function currently contains a constant for the diameter of the obstructions. This constant will become a specifiable parameter in future releases of MICE.)*

link *linker linkee link-type*
Creates a link between the linker and the linkee (where linkee can be a single agent or a list of agents). The link-type specifies the position of the linkee relative to the linker's orientation. It is assumed that, once linked, the linker has authority over the linkee, such that if the linker changes its orientation, the linkee is moved correspondingly.

unlink *linker linkee*
Removes all links between the linker and the linkee (where linkee can be a single agent or a list of agents).

send-message *agent channel-name type content &KEY (priority 0) (hearer :ALL)*
The agent sends a message through the channel *channel-name* with the *priority*. The message is of *type* with *content*. User can specify any *type* and *content* of the message as arguments of this function. If the *hearer* (an agent name) is not specified explicitly as a keyword argument, all participants of the channel including the speaker will receive messages.

recv-messages *agent channel-name &KEY (count most-positive-fixnum) (clear nil)*
The agent receives at most *count* messages from the *channel-name*. If the keyword argument *clear* is set to **t**, the remaining messages of the channel are cleared after returning at most *count* messages from the channel. If the number of messages to receive from the channel is greater than *count*, messages are selected based on the *priority* of the message. The default *count* value is infinity.

select-messages *messages &KEY (type :ALL) (status :SUCCESS)*
select-messages is an auxiliary function for convenient handling of received messages. Since the **recv-messages** function just returns a list of messages, the user may want to select only part of the messages of *type* and *status*. *type* is the user specified value as an argument of the **send-message** function and the *status* is set by MICE during transmission. The possible values for *status* are `:OVER-CAPACITY, :OUT-OF-RANGE, :OBSTRUCTED, :FAILURE` and `:SUCCESS`. The status of `:OVER-CAPACITY`, `:OUT-OF-RANGE`, `:OBSTRUCTED` and `:FAILURE` represent the reasons for the transmission failure. Especially, `:FAILURE` messages occur when the reliability value of the channel is less than 1.0. The messages successfully delivered over the channel have `:SUCCESS` status. **select-messages** returns two values using *multiple values*. The first value is the list of selected messages and the second value is the list of remaining messages after selection.

4.4 Messages

When an agent issues a **:SEND** command, MICE makes a *message* and transmits it over the channel. Thus, the **read-and-reset-received-messages** function following the **:RECV** command returns a list of messages of the structure MICE makes. The slots of the structure can be accessed using `message$` prefix to the slot name. The available slots are *speaker, hearer, type, content, priority, channel-name, time-created*, and *status*. The *speaker* and *hearer* are the names of the sender and receiver respectively. The *type, content, priority* and *channel-name* are those that are specified as parameters of the **:SEND** command. The *time-created* slot indicates the simulated time when the message is created. The *status* is the status of message delivery. A message that is successfully delivered will have `:SUCCESS` as its *status*, and returned messages will have one of `:OVER-CAPACITY`, `:OUT-OF-RANGE`, `:OBSTRUCTED`, `:FAILURE` as a status (see Section 3.2).

4.5 Examples of Invocation Functions

Example 1. Following is an example of invocation function by which the agent sends and receives a message to other agents and then moves randomly.

```
(defun communication-agent (agent)
  (let ((messages (read-and-reset-received-messages :RESET-VALUE :EMPTY)))
    (cond ((eq :EMPTY messages)
           `((:RECV channel-1)))
          (t
           ;; print received messages
           (mapcar #'(lambda (message)
                       (format t "~%Time ~a: ~a <== ~a: content ~a, status ~a"
                               *current-time*
                               (message$hearer message)
                               (message$speaker message)
                               (message$content message)
                               (message$status message)))
                   messages)
           ;; Send a message containing the sender's name and current-time.
           `((:SEND channel-1 test ,(list (agent$name agent) *current-time*))
             (:MOVE ,(nth (random 4) '(:NORTH :SOUTH :EAST :WEST)))))))))
```

Example 2. Following is an example of invocation function by which the agent interleaves scanning and moving. In moving, it randomly chooses among directions that are not blocked.

```
(defun wandering-agent (agent)
  (let ((scan-data (read-and-reset-scanned-data)))
    (if scan-data
        (let* ((agent-blockers (agent$blocked-by-types agent))
               (agent-location (agent$location agent))
               (loc-x (location$x agent-location))
               (loc-y (location$y agent-location))
               (valid-directions-and-locations
                 (list (list :EAST (make-location :X (1+ loc-x) :Y loc-y))
                       (list :WEST (make-location :X (1- loc-x) :Y loc-y))
                       (list :NORTH (make-location :X loc-x :Y (1- loc-y)))
                       (list :SOUTH (make-location :X loc-x :Y (1+ loc-y))))))
          (dolist (grid-desc scan-data)
            (let ((valid-dir-and-loc
                    (some
                      #'(lambda (valid-one)
                          (when (equalp (second valid-one)
                                        (grid-description$coordinates grid-desc))
                                valid-one))
                          valid-directions-and-locations)))
```

```
        (when (and valid-dir-and-loc
                   (some
                     #'(lambda (near-agent)
                         (member (agent$type near-agent) agent-blockers))
                    (grid-description$agents grid-desc)))
          (setf valid-directions-and-locations
                (delete valid-dir-and-loc
                        valid-directions-and-locations
                        :TEST 'equalp)))))
    `((:MOVE ,(first (nth (random (list-length
                                   valid-directions-and-locations))
                          valid-directions-and-locations)))))
  `((:SCAN :ALL)))))
```

5 MICE Execution

Having decided on a simulated environment for the agents and having implemented the agents' decisionmaking procedures, we can continue on to run MICE.

5.1 The Environment File

The information about the grid and the specific agents in an environment are stored in a file called an *environment file*. This file has the following parts (see Section 7 for a more complete example).

Simulation Data
A global variable called `*simulation-data*` points to a structure that contains information about the simulated world. In particular, it has a field called `overall-region` that specifies the region encompassed by the grid. By default, the grid has locations from 0 to 20 (inclusive) in the x and y dimensions. To modify the grid to a 10 by 10 size, the environment file would have an entry:

```
(setf (simulation-data$overall-region *simulation-data*)
      (make-region :X-MIN 0 :Y-MIN 0 :X-MAX 10 :Y-MAX 10))
```

Grid Features
Next, any features of the grid (other than agents) are defined. For example, if we wanted to put a feature at location (10,10) that would prevent agents of type `:PREY` from occupying that location, we could add a feature to the grid-element at that location that associates the key `:BLOCKED-TYPES` with the agent type `:PREY`:

```
(setf (grid-element$features
        (get-grid-element (make-location :X 10 :Y 10) t))
      (acons :BLOCKED-TYPES (list :PREY) nil))
```

Agents
Finally, we would instruct MICE to make the agents to populate the environment. For example, we might make an agent named sitting-duck of type `:PREY`, who begins in location (0,0), takes 2 time units to move in any direction, is captured by `:PREDATOR` agents, is graphically depicted as a filled square, and has an invocation-function called `prey-invocation-function`.

```
(create-agent :NAME 'SITTING-DUCK
              :TYPE :PREY
              :LOCATION (make-location :X 0 :Y 0)
              :BLOCKED-BY-TYPES (list :PREDATOR :PREY)
              :MOVE-DATA (make-move-data :NORTH 2 :SOUTH 2 :EAST 2 :WEST 2)
              :CAPTURED-BY-TYPES (list :PREDATOR)
              :DRAW-FUNCTION '(square-icon :FILLED t)
              :INVOCATION-FUNCTION #'prey-invocation-function)
```

5.2 Starting MICE

Once an environment file has been prepared, MICE can be invoked with the `mice` function:

mice *environment-file &KEY (time-limit 200)*
Initiates a MICE run. Reads in the specified *environment-file*, and then simulates the actions of the agents. The *time-limit* keyword argument indicates the maximum simulated time that is allowed.

5.3 MICE Activities

At any given time, MICE invokes the agent with the least advanced simulated clock. If there is a tie, it will by default give preference to the agent with the lowest authority. The predicate used to sort agents for execution is user-modifiable, and is bound to the global variable `*sort-agent-predicate*`.

After it executes an agent, MICE checks to see what simulated time the agent with the least-advanced clock is at, and if this is greater than it was before the agent executed (MICE maintains a "global" clock to store this value), then MICE must resolve any conflicting actions that might have occurred between the previous value of the global clock and the current global clock time. It steps through the intervening times, and for each time:

1. It checks any actions initiated by the agents at that time and the resulting state of the agents as a consequence of those actions.
2. It resolves any conflicts between the actions (see Section 3.4).
3. It checks the resolved situation against the set of possible agent interactions. For example, it checks the predicates for removing agents, such as when a `:PREY` agent is surrounded by `:PREDATOR` agents.
4. It takes any actions triggered by the situation.
5. If any actions were taken in step 4, it goes back to step 3. Otherwise, it is done.

5.4 MICE Termination

The criteria for termination are user-specified. Sometimes we want MICE to stop when no agent has moved for some fixed amount of time. Other times we want MICE to stop when no goals are left to achieve (such as when all of the `:PREDATOR` agents have captured all of the `:PREY` agents). Currently, the user defines a predicate called `mice-continue-p` that specifies the conditions under which MICE should continue the simulation. By default, the function continues until either the time-limit has been reached or until no agent has moved.

```
-------------- TIME 0 --------------
. . R H H R . . .
. . J J J J . . .
1 8 4 9 . . . . .
R r R C c C T t T
r s r c s c t s t
. R C R T C R C .
```

Figure 3: Text output of example icons found in testicons.env.

6 User Interface

6.1 Graphics

The graphical icons which MICE uses to represent agents and grid features are drawn using a set of device-independent graphics calls, which then are translated into the appropriate device-dependent calls based on the "graphics mode". We have written the device-dependent routines for X-windows, the Macintosh, and the TI Explorers. We also have a set of Postscript routines that can produce accurate Postscript depictions of MICE simulation grids, so that printing MICE grids or including grid images in LaTeX documents is simple (as seen in Figure 2 on Page 11). The `di-set-graphics-mode` function sets the graphics mode to one of the following symbols: `'X`, `'TI`, `'MAC`, `'PS`.

The default width of the graphics display is held in the global variable `*display-width*`. The size of the display window can be changed by setting this variable (in approximate inches) before running MICE. The Macintosh version of MICE allows resizing of the window during execution by dragging the corner of the display window.

If no graphics display is available, a simple textual representation of the MICE grid can be obtained by setting `*graphics?*` to **nil**. MICE will then print to the screen a matrix with dots (.s) in empty grid locations and single characters in filled locations. The single character will be the first character of a `:LABEL` that is specified in the draw-function call, or a character representing the icon function (e.g., "R" for a rectangle-icon). Figure 3 shows the same simulation grid as Figure 2, in the textual representation.

If you wish to turn off all forms of the MICE grid display, for example while running multiple simulations (more quickly) in the background, set `*display?*` to **nil**.

To save the PostScript code for a single grid image, use:

save-grid-ps *&key (time *current-time*) (file "<time>.lps"))*
This creates a postscript file that can be printed directly, or included into a LaTeX document using the psfig macros.

6.2 Saving and Restoring Runs

Experimental runs can be saved to a file and later restored and redisplayed. MICE offers two options for saving, restoring and redisplaying runs: one option saves the device independent graphics calls needed to reproduce the graphical description of a simulation run, the other saves MICE data structures including the entire state history for each agent.

save-mice-graphics *file*
This function saves the device independent graphics calls needed to reproduce the graphical depiction of a MICE simulation run. Note that even if the graphic display is disabled by setting `*display?*` to **nil**, the graphics calls are still recorded so that, when running multiple simulations in the background, you can have a program detect interesting results and save the simulation for later examination. However, if you set `*graphics?*` to **nil**, the device-independent graphics calls are never made, and thus cannot be saved with this option.

save-mice-history *file*
This function saves the MICE data structures into *file* including the definition of each agent, the features of the grid, and the history of commands executed, locations occupied, links created, and so on.

restore-mice-graphics *file*
This function loads the device independent graphics calls contained in *file* so that they can be redisplayed by calling **redisplay-graphics**.

restore-mice-history *file*
This function loads the data structures contained in *file* so that they can either be inspected or redisplayed by calling **redisplay-history**.

redisplay-graphics *&key (start 0) end (sleep-time 0) (breaks nil)*
This function redisplays a MICE simulation run using the set of device independent graphics routines that were either just executed or just restored. The display will begin at the specified `:START` time and end at the specified `:END` time (or at the end of the simulation run). If the simulation is being redisplayed too quickly, use `:SLEEP-TIME` to set the number of seconds to pause between each time step. Use `:BREAKS` to specify a list of times at which the redisplay should pause, waiting for a keystroke to continue.

redisplay-history *&key (start 0) end (sleep-time 0) (breaks nil)*
This function redisplays a MICE simulation run using the MICE data structures from the run just executed or just restored. The parameters are identical to those of **redisplay-graphics** above.

6.3 Statistical Measures

Currently, MICE can optionally maintain a small number of statistics, corresponding to the number of moves that agents attempt and the number of these that are successful (do not lead to conflict). More types of measuring will be added in future releases.

6.4 Simulating Real-Time

MICE has a variable called `*real-time-knob*` that can be used as a simple mechanism for rewarding (penalizing) agents for making decisions quickly (slowly). By default, the `*real-time-knob*` is **nil**, which disables it. If it is non-nil, it should contain a number. This number is used by MICE to map actual runtime into simulated time units.

Specifically, MICE records the time (using *get-internal-real-time*) when the invocation-function for an agent is called and when it returns, and computes the total elapsed time for invocation in seconds. This value is multiplied by the value of `*real-time-knob*`, and the result is rounded

to an integer value. MICE then inserts, onto the front of the list of commands returned by the invocation function, a **:REASONING** command for the integer value computed. So, for example, if an agent spends 5.4 seconds generating commands for MICE, and `*real-time-knob*` is set to 0.5, then the commands the agent has generated are postponed by 3 simulated time units.

6.5 Debugging

MICE has a few variables that can be used to help in debugging. These include:

`*verbose?*` If non-nil, MICE sends to text output an indication of what command each agent is sending to MICE.

`*debug?*` If non-nil, MICE announces each invocation of an agent.

`*collision-verbose*` If non-nil, MICE informs the user when two agents have collided.

`*link-verbose*` If non-nil, MICE informs the user when a linking activity has been attempted.

7 Implementation Examples

In the following subsections, we illustrate through very simple examples how the MICE environment, domain predicates, and agents can be defined. Users are referred to additional examples accompanying the MICE system.

7.1 Example Environment File

This file gives an example of a simple environment where four predators attempt to capture one prey agent. To make matters more interesting, there is a wall running north-south in the middle of the grid that blocks predators but does not block prey.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;        SIMPLE PREDATOR-PREY ENVIRONMENT
;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; First, modify the grid to represent a north-south wall in the middle of
; the grid.  This wall blocks out predators, but prey can move through it

(setf (grid-element$features (get-grid-element (make-location :x 10 :y 8) t))
 (setf (grid-element$features (get-grid-element (make-location :x 10 :y 9) t))
  (setf (grid-element$features (get-grid-element (make-location :x 10 :y 10) t))
   (setf (grid-element$features (get-grid-element (make-location :x 10 :y 11) t))
    (setf (grid-element$features (get-grid-element (make-location :x 10 :y 12) t))
          (acons :BLOCKED-TYPES (list :PREDATOR) nil)))))))

; Also, set the draw-function for these grid locations to be square icons

(setf (grid-element$draw-function (get-grid-element
                                    (make-location :x 10 :y 8) t))
 (setf (grid-element$draw-function (get-grid-element
                                     (make-location :x 10 :y 9) t))
  (setf (grid-element$draw-function (get-grid-element
                                      (make-location :x 10 :y 10) t))
   (setf (grid-element$draw-function (get-grid-element
                                       (make-location :x 10 :y 11) t))
    (setf (grid-element$draw-function (get-grid-element
                                        (make-location :x 10 :y 12) t))
          '(square-icon)))))))

; Next, create a communication channel

(create-channel :NAME 'channel-1
                :AGENTS '(PRED1 PRED2 PRED3 PRED4)
                :DELAY 1
                :CAPACITY nil
                :RELIABILITY 1.0
                :RANGE NIL
                :ORIENTATION-SENSITIVE-P :UNKNOWN
                :OBSTRUCTED-BY nil)

; Now, create a prey agent.  This agent begins in the center of the
; environment.  It can be captured by predators and cannot capture
; anything.  It's REMOVE-P predicate indicates that it should be removed
; from the environment when it is captured, and its REMOVE-FUNCTION
; simply returns its own name as the agent to remove.  Its DRAW-FUNCTION
```

```
; gives it a square clock-face representation with the label "P".
; It is blocked by (cannot share a location with or pass through) other
; prey and predator agents, it requires 2 simulated time units to move in
; any direction, and it has limited sensors that only sense locations at
; most 2 away from the agent in all directions (i.e. it senses the 5 by 5
; region around and including its location).

(create-agent :NAME 'PREY1
              :TYPE :PREY
              :LOCATION (make-location :X 10 :Y 10)
              :CAPTURED-BY-TYPES '(:PREDATOR)
              :CAPTURE-TYPES '()
              :REMOVE-P 'captured-agent-p
              :REMOVE-FUNCTION 'mice-self
              :COLLISION-FUNCTION 'collision-function
              :DRAW-FUNCTION '(rectangle-icon :label "P")
              :BLOCKED-BY-TYPES (list :PREDATOR :PREY)
              :MOVE-DATA (make-move-data :NORTH 2 :SOUTH 2 :EAST 2 :WEST 2)
              :INVOCATION-FUNCTION #'prey-invocation-function
              :SENSORS
               (list (make-sensor-data
                       :RANGE (make-region :X-MIN -2 :Y-MIN -2
                                           :X-MAX 2 :Y-MAX 2))))

; Finally, create four predator agents.  Each agent begins at a different
; corner of the environment.  Each can capture prey and cannot be captured.
; Each has a DRAW-FUNCTION that represents predator i as a round clock-face
; with a hand pointing to time i.  Each is blocked by (cannot share a location
; with or pass through) prey and other predator agents. Each requires 1
; simulated time unit to move in any direction, and has sensors that sense
; locations at most 10 away from the agent in all directions.

(create-agent :NAME 'PREDATOR1
              :TYPE :PREDATOR
              :LOCATION (make-location :X 0 :Y 0)
              :CAPTURED-BY-TYPES '()
              :CAPTURE-TYPES '(:PREY)
              :COLLISION-FUNCTION 'collision-function
              :DRAW-FUNCTION '(clock-icon :hour 1)
              :BLOCKED-BY-TYPES (list :PREDATOR :PREY)
              :MOVE-DATA (make-move-data :NORTH 1 :SOUTH 1 :EAST 1 :WEST 1)
              :INVOCATION-FUNCTION #'predator-invocation-function
              :CHANNELS '(channel-1)
              :SENSORS
               (list (make-sensor-data
                       :RANGE (make-region :X-MIN -10 :Y-MIN -10
                                           :X-MAX 10 :Y-MAX 10))))

(create-agent :NAME 'PREDATOR2
              :TYPE :PREDATOR
              :LOCATION (make-location :X 20 :Y 0)
              :CAPTURED-BY-TYPES '()
              :CAPTURE-TYPES '(:PREY)
              :COLLISION-FUNCTION 'collision-function
```

```
               :DRAW-FUNCTION '(clock-icon :hour 2)
               :BLOCKED-BY-TYPES (list :PREDATOR :PREY)
               :MOVE-DATA (make-move-data :NORTH 1 :SOUTH 1 :EAST 1 :WEST 1)
               :INVOCATION-FUNCTION #'predator-invocation-function
               :CHANNELS '(channel-1)
               :SENSORS
                (list (make-sensor-data
                        :RANGE (make-region :X-MIN -10 :Y-MIN -10
                                            :X-MAX 10 :Y-MAX 10))))

(create-agent :NAME 'PREDATOR3
               :TYPE :PREDATOR
               :LOCATION (make-location :X 0 :Y 20)
               :CAPTURED-BY-TYPES '()
               :CAPTURE-TYPES '(:PREY)
               :COLLISION-FUNCTION 'collision-function
               :DRAW-FUNCTION '(clock-icon :hour 3)
               :BLOCKED-BY-TYPES (list :PREDATOR :PREY)
               :MOVE-DATA (make-move-data :NORTH 1 :SOUTH 1 :EAST 1 :WEST 1)
               :INVOCATION-FUNCTION #'predator-invocation-function
               :CHANNELS '(channel-1)
               :SENSORS
                (list (make-sensor-data
                        :RANGE (make-region :X-MIN -10 :Y-MIN -10
                                            :X-MAX 10 :Y-MAX 10))))

(create-agent :NAME 'PREDATOR4
               :TYPE :PREDATOR
               :LOCATION (make-location :X 20 :Y 20)
               :CAPTURED-BY-TYPES '()
               :CAPTURE-TYPES '(:PREY)
               :COLLISION-FUNCTION 'collision-function
               :DRAW-FUNCTION '(clock-icon :hour 4)
               :BLOCKED-BY-TYPES (list :PREDATOR :PREY)
               :MOVE-DATA (make-move-data :NORTH 1 :SOUTH 1 :EAST 1 :WEST 1)
               :INVOCATION-FUNCTION #'predator-invocation-function
               :CHANNELS '(channel-1)
               :SENSORS
                (list (make-sensor-data
                        :RANGE (make-region :X-MIN -10 :Y-MIN -10
                                            :X-MAX 10 :Y-MAX 10))))

;;;  End of Environment File
```

7.2 Example Domain Predicates

Below are some predicates used in the simple predator/prey environment described in the environment file.

```
;;; -----------------------------------------------------------------------
;
; Returns non-nil if the agent is surrounded by agents by which this agent
; can be captured.  Agent is an agent data-structure; time is the agent's
; current time. *agent-schedule-queue* is the list of currently active
; agents maintained by MICE.

(defun captured-agent-p (agent time)
  (let ((surrounding-agents
         (find-neighboring-agents agent *agent-schedule-queue* time)))
    (and (>= (length surrounding-agents) 4)
         (every
           #'(lambda (direction)
               (let ((new-location
                       (compute-new-location (agent$location agent) direction)))
                 (some #'(lambda (a)
                           (and (equalp (agent$location a) new-location)
                                (member (agent$type a)
                                        (agent$captured-by-types agent))))
                       surrounding-agents)))
           '(:NORTH :SOUTH :EAST :WEST)))))

;;; -----------------------------------------------------------------------
;
; Returns a list of all of the agents in the others argument that are in
; locations adjacent to the given agent at the specified time.

(defun find-neighboring-agents (agent others time)
  (cond ((null others) nil)
        ((= (+ (abs (- (location$x (find-agent-location agent time))
                       (location$x (find-agent-location (first others) time))))
               (abs (- (location$y (find-agent-location agent time))
                       (location$y (find-agent-location (first others) time)))))
            1)
         (cons (first others) (find-neighboring-agents agent (rest others) time)))
        (t (find-neighboring-agents agent (rest others) time))))

;;; -----------------------------------------------------------------------
;
; Moves agents that have collided and block each other back to their
; previous locations.

(defun collision-function (agent1 agent2 time)

  "COLLISION-FUNCTION agent1 agent2 time

Assumes that agent1 and agent2 have passed through each other or are
attempting to occupy the same location.  If the agents block each other,
```

```
moves them back to where they were at previous time, otherwise leaves them
alone."

  (when (or (member (agent$type agent1) (agent$blocked-by-types agent2))
            (member (agent$type agent2) (agent$blocked-by-types agent1)))
    ; find which of the agents moved to cause the problem (possibly both)
    (let ((conflict-agents
            (mapcan #'(lambda (agent)
                        (when (not (equalp (find-agent-location agent time)
                                           (find-agent-location agent
                                                                (1- time))))
                          (list agent))))))
      ; move each of the agents to where it was at the previous time
      (move-agents-to-location-at-given-time conflict-agents (1- time))
      t)))
```

7.3 Example Agent Implementation

An agent's :INVOCATION-FUNCTION can be any lisp-callable function. For AI research, the function should employ AI techniques to make appropriate decisions. Because such functionality can be very complex, we instead here present a function that simply queries the user for some action, and then passes the command back to MICE in the proper format. More complex examples are bundled with the code, including a complete example of a predator-prey scenario in which the predators negotiate using the Contract-Net protocol.

```
;---------------------------------------------------------------------------
;
; Human-agent is an invocation-function that simply queries the user for some
; action and returns the appropriate command in the proper format for MICE.

(defun human-agent (agent)
  (format t "~%Select command for agent ~a~%" (agent$name agent))
  (format t "Move, Link, Unlink, Scan, Reason, sTop, Direction or Quiescent ")
  (let ((response (char-upcase (read-char))))
    (cond
      ((eql response #\M)
       (format t "~%Select direction to move (N, S, E, W). ")
       (setf response (char-upcase (read-char)))
       (format t "~a~%" response)
       (cond ((eql response #\N)
              '((:MOVE :NORTH)))
             ((eql response #\S)
              '((:MOVE :SOUTH)))
             ((eql response #\E)
              '((:MOVE :EAST)))
             ((eql response #\W)
              '((:MOVE :WEST)))
             (t '((:MOVE nil)))))

      ((eql response #\L)
       (format t
          "~%Select link type: Front, Left, Right, Back, Next-to or Shared-loc. ")
       (let* ((response (char-upcase (read-char)))
              (link-type (case response
                           (#\F :FRONT)
                           (#\L :LEFT)
                           (#\R :RIGHT)
                           (#\B :BACK)
                           (#\N :NEXT-TO)
                           (#\S :SHARED-LOC)
                           (otherwise (cerror "Default to :NEXT-TO."
                                              "Illegal link type selected.")
                                      :NEXT-TO))))
         (format t "~%Enter agent to create ~a link with. " link-type)
         (setf response (string-upcase (read-line)))
         '((:LINK ,(intern response) ,link-type))))

      ((eql response #\U)
       (format t "~%Enter agent to unlink from. ")
```

```
 (setf response (read-line))
 `((:UNLINK ,(intern (string-upcase response)))))

((eql response #\S)
 (format t "~%Enter time spent scanning. ")
 (setf response (read-line))
 `((:SCAN ,(string-to-int response))))

((eql response #\R)
 (format t "~%Enter time spent reasoning. ")
 (setf response (read-line))
 `((:REASONING ,(string-to-int response))))

((eql response #\T)
 (format t "~%Agent terminating.~%")
 :STOP)

((eql response #\Q)
 (format t "~%Agent quiescent.~%")
 :QUIESCENT)

((eql response #\D)
 (format t "~%Enter direction of rotation. (Right, Left) ")
 (let ((direction (if (eql (char-upcase (read-char)) #\L) :LEFT :RIGHT)))
   (format t "~%Enter number of quadrants to rotate ~a. " direction)
   `((:ROTATE ,direction ,(- (read-char) (int-char #\0))))))

(t (format t "~%Illegal selection.~%"))))
```