RoboCup Rescue: Agent Development Kit
Version 0.4

Michael Bowling*
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3890
mhb@cs.cmu.edu

December 5, 2000

Contents

*Currently a visiting researcher at Kitano Symbiotic Systems Project, ERATO, JST in Tokyo, Japan.

1 Introduction

RoboCup Rescue is a new initiative designed to foster multiagent research in a socially-significant domain. This document describes an Agent Development Kit (ADK) to help ease the burden of designing agents for this new domain. Although, one of the goals of the ADK is to abstract away many of the gritty details of the simulation environment, it is still recommended that you are familiar with the RoboCup Rescue Simulator Manual, available from http://kiyosu.isc.chubu.ac.jp/robocup/Rescue/manual-English-v0r3/.

1.1 Purpose

The purpose of the RoboCup Rescue ADK is to simplify the procedure for developing interesting agents for the simulation system. This purpose is two-fold. First, to abstract away the details of the agent-kernel communication protocol so agent developers can focus on agent development. And second, to provide a toolbox for building high-level behaviors.

These goals are to be achieved while providing maximum flexibility to the agent developers. It is an important goal of the ADK not be biased towards any particular solution to the agent development problems. The toolbox should only contain well-known tools for addressing only the simplest problems inherent in the rescue environment. For example, a search algorithm for finding shortest paths is a basic tool that this kit includes, but an implementation of a specific agent communication protocol is not, since this is an area of ongoing research.

Currently the ADK is only available in C++.

1.2 Organization

The ADK consists of three components. The organization is shown in Figure 1. The first component is the simulator codebase that is provided with the RoboCup rescue simulation environment. The second component is the Controller class which provides an abstraction for the agent's interaction with the kernel. The third component is a Memory class that encapsulates the agent's current view of the environment.

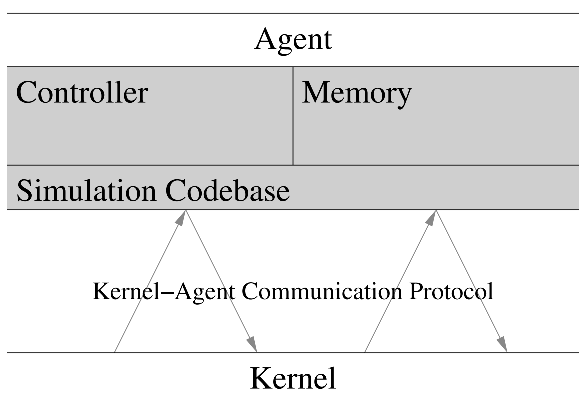

Figure 1: The organization of the Agent Development Kit. The shaded regions correspond to the components of the ADK. The Simulator Codebase, Controller, and Memory are described in Sections 2, 3, and 4, respectively.

Although, the Memory and Controller components depend on the simulator codebase they do not depend on eachother. This allows agent developers to use their own Memory component, or one derived from the ADK, and still make use of the ADK's Controller component. It's also possible to use the Memory component independent of the Controller, though this is probably unlikely.

Each of these components will be described in detail in the following sections. The final section will contain an example agent built using the ADK to help illustrate how these components work together.

2 Simulator Codebase

The available code that accompanies the RoboCup Rescue simulation provides some crucial support, which the ADK makes use of. Agents built using the ADK will often have to work with the data types they provide. This code performs a variety of functions: network communication, configuration file support, and most importantly an object hierarchy defining the various objects in the rescue environment.

For the most part, agent developers do not need to be familiar with the communication or configuration file support. On the other hand, agent developers will have to be very familiar with the object hierarchy. Currently there is no documentation describing the specifics of the object hierarchy, although the RoboCup Rescue Simulation Manual does provide some high-level description. Until there is documentation, it is best to simply consult the "`objdef.h`" file and example agents. Figure 2 shows the class hierarchy defined in the codebase, which may be of some assistance when consulting "`objdef.h`".

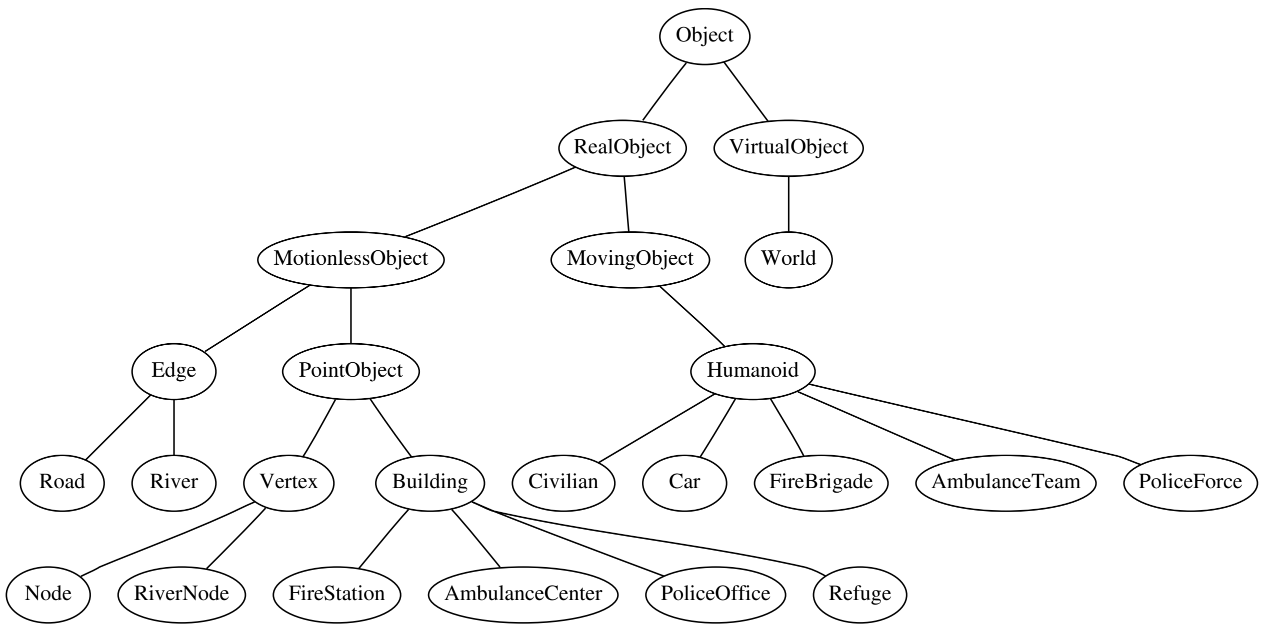

Figure 2: The class hierarchy defined in "`objdef.h`".

The codebase also defines a class called `ObjectPool`, that is essentially a collection of `Objects`. The Controller class described in Section 3 requires an object of this type to be used as its memory. This allows agent developers to use either a simple `ObjectPool`, the Memory class described in Section 4, or their own class derived from `ObjectPool`.

3 Controller Class

`Controller` is the base class from which all agents should be derived. It has the following features:

- Supports an ObjectPool (or derived class) as memory.
- Supports multiple agents sharing the same communication socket. Improves performance by only requesting the initial state of the world once for all agents.
- Derived controllers need only implement the `act` method. They may also optionally implement the `init`, `sense_changed`, and `hear` methods.
- Provides support for multiple modes of operation (e.g. continuous acting, acting after each sensation.) See `m_act_policy` field in Section 3.2.1.
- Simplifies the execution of actions with convenience functions for all the primitive actions.

Controller.hxx

```
namespace Rescue {

class Controller
{
```

3.1 Public Methods

These methods are basically for `main` to create, connect, and start agents running. See Section 5.1 for an example implementation of `main`.

Controller.hxx

```
  static void setup_connection(char *host, int port);
```

All agents share the same UDP connection to the kernel. `setup_connection` is a static method to setup that initial connection for all the agents. This must be called once before any `Controller`'s constructor gets called.

Controller.hxx

```
  Controller(int type, ObjectPool *memory);
  virtual ~Controller();
```

The constructor takes a type parameter which is a bitwise-or of the agent types that this agent can assume. For example,

```
Controller *c = new Controller((1 << AGENTTYPE_POLICE_FORCE) |
                               (1 << AGENTTYPE_POLICE_OFFICE),
                               new ObjectPool());
```

The second argument is the `ObjectPool` or object derived from that class which is used as the Agent's memory. Although, not in line with the spirit of a distributed environment, it's also possible to pass the same ObjectPool to multiple agents so that they have a single shared memory. This may improve performance for non-critical agents, such as civilians.

Controller.hxx

```
  bool connect();
  static void go();
```

The `connect` method contacts the kernel to connect into the simulation environment, and will return `true` if successful. After all agents have been created and connected to the kernel, the static method `go` will start the agents running.

3.2 Protected Fields and Methods

Agents are derived classes and so therefore have access to these fields and methods. They fall into the following categories:

- Basic Support,
- Memory Interface,
- Primitive Actions, and
- Extendibility.

3.2.1 Basic Support

These are the basic methods that derived agents can override to control their behavior. Agents are not required to override any of these methods, but will almost certainly want to override the `act` method.

Controller.hxx
```
  virtual void init() {};
  virtual void sense_change(Object *o) {};
  virtual void sense_object(S32 time, Object *o, Input &input);
  virtual void hear(Object *sender, const char *msg) {};
  virtual void act() { send(); }

  enum ActPolicy { ACT_ONCE_WHEN_IDLE, ACT_ONCE_PER_SENSE, ACT_WHEN_IDLE,
                   ACT_LATER, ACT_NEVER };
  ActPolicy m_act_policy;
```

The `init` method gets called immediately after the agent connects to the kernel and is sent the GIS data that comprises the initial state of the world. By default this does nothing.

The `sense_change` method gets called when a new sensation from the kernel is given for a specific object. This method is called after the object has been updated with the property changes.

The `sense_object` method is useful if an agent needs to know how a property changed (i.e. both its value before and after updating) when a sensation is received. This method must be used with some care, though. **The `Controller`'s definition of this method must be called at some point within any method that overrides it.** The structure of any overriding method should look like:

```
void Agent::sense_object(S32 time, Object *o, Input &input)
{
  // Examine the current value of the object, storing information
  // in the local variables

  Controller::sense_object(time, o, input);

  // Examine new values of the object and compare to old values.
}
```

Section 5.2.3 gives an example of how this method can be used to notify others of crucial observations.

The `hear` method is called when another agent communicates with this agent.

The `act` method is the primary method that agents interact with the environment. This method should execute actions (see Section 3.2.3), which will be sent to the kernel to be executed. By default this does nothing besides sending any currently pending actions. Derived classes should be sure to call `Controller::act()` after specifying actions to be sure they are sent to the kernel. The `act` method should exit quickly.

A number of models of execution can be specified with `m_act_policy`, which should be set in a derived class' constructor. The description of how these policies affect when the `act` method is invoked is described in Table 1. The default setting is `ACT_ONCE_WHEN_IDLE`, but can be changed in the constructor of a derived class.

Controller.hxx
```
  void send();

  virtual void sense(S32 time, Input &input);
```

Value	Calling Policy
ACT_ONCE_PER_SENSE	act is called once after a KA_SENSE message is received from the kernel.
ACT_WHEN_IDLE	act is called whenever there are no messages from the kernel to be processed. act may be called any number of times (including none) between KA_SENSE messages.
ACT_ONCE_WHEN_IDLE	Like ACT_WHEN_IDLE, except act will not be called more than once between KA_SENSE messages from the kernel.
ACT_LATER	act will not get called while this is the value of m_act_policy. KA_SENSE and KA_HEAR messages are still processed. This is useful for agents that are awaiting some message or change in state before executing actions.
ACT_NEVER	Like ACT_LATER but KA_SENSE and KA_HEAR messages are not processed. The agent will not perform any actions for the rest of the simulation.

Table 1: The effect of the m_act_policy field on when act is called.

The action methods in Section 3.2.3 only queue commands to be sent to the kernel. The send method can be invoked to flush any pending commands to the kernel. This may be useful if methods besides act (whose default implementation invokes send()) execute actions that should be sent to the kernel immediately. This method cannot be overridden.

The sense method is included for completeness. Derived classes should not need to override this method. This method receives the raw input from the kernel for a sense action and callse sense_object for each object for which information is received. It also updates the current time.

3.2.2 Memory Interface

These are some basic methods and fields that agents can use to query their view of the world. The Memory class described in Section 4 provides more useful and advanced methods.

Controller.hxx

```
ObjectPool *m_memory;

template <class T>
T *lookup(Id id) { return dynamic_cast<T *>(m_memory->get(id)); }

template <class T>
T *is_a(Object *o) { return dynamic_cast<T *>(o); }
```

m_memory is the ObjectPool or derived class that was given in the agent's constructor. It can be queried for information about the current view of the world. The lookup<class> method is a convenience function to ask the memory for the object representing a specific ID. The template parameter specifies the expected type, which is checked. The is_a<class> method is a convenience function that checks whether an object is of a specific type. If not, it returns NULL. For example,

```
Road *road = lookup<Road>(road_id);
Building *building = is_a<Building>(object);

if (building) { ... }
```

Controller.hxx

```
S32 m_time;
S32 m_id;
MovingObject *m_self;
```

These are useful fields related to the agent's memory. m_time is the time of the last sensation that was incorporated into the memory model. m_id is the ID of the agent. m_self is the object representing the agent, itself.

3.2.3 Primitive Actions

These methods are the primitive actions available to agents. Each method queues a message to be sent to the kernel to execute the action. All queued actions will then be sent when send is called, which is the default action of act (see Section 3.2.1). Most of these actions are self-explanatory, though a few clarifications are made below.

Controller.hxx

```
void act_move(Object **path); // path must be a NULL terminated array.
void act_open(Road *target);
void act_rescue(Humanoid *target);
void act_load(Humanoid *target);
void act_unload();

void act_say(Object *target, const char *msg);
void act_tell(Object *target, const char *msg);

void act_say_list(Object **targets, const char *msg);
void act_tell_list(Object **targets, const char *msg);
void act_extinguish_simple(Building *target);

void act_extinguish(); // Unimplemented
```

The act_move method requires an array of Object pointers. This array is the path of neighboring Objects that the agent is to traverse. The array must be NULL terminated but need not remain valid (i.e. it can be deallocated) after calling the method. See Section 4.2.2 for a function that will generate useful paths for this method.

The act_say_list and act_tell_list are convenient methods for communicating a message to multiple agents. It is equivalent to calling act_say or act_tell for each agent in the array. Like the array parameter of act_move the target array must be NULL terminated.

The act_extinguish method is currently not implemented since it's unclear what the interface to this primitive action should be. The act_extinguish_simple is a simpler version that provides reasonable defaults for the parameters to extinguish the specified target.

3.2.4 Extendibility

These methods and fields are not likely to be useful for most agent implementations but are still provided to insure maximum extendibility and flexibility. They should only be used with great consideration.

Controller.hxx

```
typedef std::map<Id, AutoPtr<Controller> > IdMap;
static IdMap m_controllers;

static Address m_address;
static LongUDPSocket m_socket;

int m_type;

Output m_output;
Output::Cursor *m_output_base;
void packet_start(Output &output, Header header);
void packet_end(Output &output);
void packet_send(Output &output);
```

4 Memory Class

The goal of the Memory class is to provide a simple and easy interface for querying details about the current view of the environment. The class is derived from `ObjectPool` and so can be used as the memory object for the `Controller` class. In addition to the collection of objects it provides a number of powerful query mechanisms.

Memory.hxx
```
namespace Rescue {

class Memory : public ObjectPool
{
```

4.1 Datatypes

The query mechanisms used by the `Memory` class strives to be as general as possible. In order to achieve this goal it makes use of a number of data structures that are used in the interfaces of the various methods.

4.1.1 Object Functions

An object function is a class that implements a mapping from `Objects` to `doubles`. Object functions must be derived from the pure class `ObjectFn`.

Memory.hxx
```
  struct ObjectFn {
    virtual bool consider(const Object *o) const = 0;
    virtual double f(const Object *o) const = 0;
  };
```

The `consider` method is a boolean function that returns `true` if its argument should even be considered at all (i.e. if the object is the domain of the function). The `f` method is the actual mapping.

Predefined Object Functions A number of predefined object functions are provided. Each of these are explained below. Some of these are used as the defaults to later `Memory` methods. Others are generally useful for implementing other object functions.

`UniformObjectFn` is a trivial function which maps all objects to the value zero.

`ObjectObjectFn` is derived from `UniformObjectFn` and restricts the domain of the function to a single object.

`ClassObjectFn` is a template class derived from `UniformObjectFn` that restricts the domain of the function to `Objects` of the class specfied in the template parameter.

`DistanceObjectFn` returns a value based on Euclidean distance of the object to some specfied point. The value returned is the square of the distance in GIS units (millimeters).

`AddObjectFn` combines two `ObjectFns`. Its domain consists of the intersection of the domains of its arguments, and the value returned to be the sum.

`NotObjectFn` is derived from `UniformObjectFn` but restricts the domain to only objects *not* considered by the `ObjectFn` specified in the constructor.

The gritty details of these classes and their constructors are given below.

Memory.hxx
```
  struct UniformObjectFn : public ObjectFn {
    virtual bool consider(const Object *o) const { return 1; }
    virtual double f(const Object *o) const { return 0.0; }
  };

  struct IsObjectObjectFn : public UniformObjectFn {
    Id m_id;
    virtual bool consider(const Object *o) const { return o->id() == m_id; }
```

```
  IsObjectObjectFn(Id id) { m_id = id; }
  IsObjectObjectFn(Object *o) { m_id = o->id(); }
};

template <class T>
struct IsClassObjectFn : public UniformObjectFn {
  virtual bool consider(const Object *o) const {
    return (dynamic_cast<const T*>(o)); }
};

struct IsAgentObjectFn : public Memory::UniformObjectFn {
  virtual bool consider(const Object *o) const {
    switch(o->type()) {
    case TYPE_FIRE_BRIGADE: case TYPE_FIRE_STATION:
    case TYPE_AMBULANCE_TEAM: case TYPE_AMBULANCE_CENTER:
    case TYPE_POLICE_FORCE: case TYPE_POLICE_OFFICE:
      return true;
    default:
      return false;
    }
  }
};

struct DistanceObjectFn : public ObjectFn {
  double m_x, m_y;
  virtual bool consider(const Object *o) const { return 1; }
  virtual double f(const Object *o) const {
    double x = o->x(); double y = o->y();
    return ((x - m_x) * (x - m_x) + (y - m_y) * (y - m_y));
  }
  DistanceObjectFn(double x, double y) { m_x = x; m_y = y; }
  DistanceObjectFn(const Object *o) { m_x = o->x(); m_y = o->y(); }
};

struct AddObjectFn : public ObjectFn {
  const ObjectFn &m_a, &m_b; double m_wa, m_wb;
  virtual bool consider(const Object *o) const {
    return (m_a.consider(o) && m_b.consider(o)); }
  virtual double f(const Object *o) const {
    return (m_a.f(o) * m_wa + m_b.f(o) * m_wb); }
  AddObjectFn(const ObjectFn &a, const ObjectFn &b,
              double wa = 1.0, double wb = 1.0) : m_a(a), m_b(b) {
    m_wa = wa; m_wb = wb;
  }
};

struct NotObjectFn : public ObjectFn {
  const ObjectFn &m_a;
  virtual bool consider(const Object *o) const { return ! m_a.consider(o); }
  virtual double f(const Object *o) const { return 0.0; }
  NotObjectFn(const ObjectFn &a) : m_a(a) { }
};
```

4.1.2 Object Pair Functions

Object pair functions are similar to object functions but define a mapping from *pairs* of `Objects` to `doubles`. Only one predefined object pair function is provided.

`TraversalObjectFn` returns a value which is the distance an agent would travel in moving from object `a` from object `b` according to the GIS data. The domain is restricted to only neighboring object pairs and those which an agent could mean between (i.e. it excludes roads that cannot be travelled due to debris.)

The class definitions are below.

Memory.hxx

```
struct ObjectPairFn {
  virtual bool consider(const Object *a, const Object *b) const = 0;
  virtual double f(const Object *a, const Object *b) const = 0;
};

struct TraversalObjectPairFn : public ObjectPairFn {
```

```
  virtual bool consider(const Object *to, const Object *from) const {
    if (to->type() == TYPE_ROAD) {
      Road *r = (Road *) to;
      return (min(r->linesToHead(), r->linesToTail()) -
              floor((0.5 * (r->linesToHead() + r->linesToTail()) *
                     (r->block() / (double) r->width())) + 0.5) > 0);
    } else return 1;
  }
  virtual double f(const Object *to, const Object *from) const {
    switch(to->type()) {
    case TYPE_ROAD: return ((Road *) to)->length();
    default: return 1;
    }
  }
};
```

4.1.3 Other Datatypes

Other datatypes are also used to provide a more general interface for querying the state of the world.

Memory.hxx

```
struct Location {
  Object *m_o;
  S32 m_dist;

  Location(Object *o, S32 dist = 0) { m_o = o; m_dist = dist; }
  Location(MovingObject *o) {
    m_o = o->position(); m_dist = o->positionExtra();
  }
};

enum FunctionOrder {FUNCTION_MINIMIZE = 0, FUNCTION_MAXIMIZE = 1};
```

`Location` is most useful for specifying an exact location in the world. This is either just an object, or an object and an extra position parameter. The position parameter is used in determining the exact position along a road, according to the `positionExtra` property as described in the RoboCup Rescue Simulation Manual.

`FunctionOrder` is an enumeration, which determines wether the value, returned by by an object (pair) function should be minimized or maximized. This is used as an argument to the `find` and `find_list` methods described below.

4.2 Methods

The methods are designed to provide a useful and powerful interface for querying the agent's current view of the rescue environment. The simplest method, `Object *get(Id id)`, is inherited from the `ObjectPool` class and can be used to retrieve an `Object` of a particular ID. If using the `Controller` class a type-checking version of this method may be more useful (See Section 3.2.2.) The additional methods provided by the Memory class can be grouped into those that find objects, and those that find paths.

4.2.1 Finding Objects

These methods will find particular objects as determined by an object function.

Memory.hxx

```
Object *find(const ObjectFn &q, FunctionOrder order = FUNCTION_MAXIMIZE);

Object **find_list(const ObjectFn &q, double threshold = -MAXDOUBLE,
                   FunctionOrder order = FUNCTION_MAXIMIZE);
```

The `find` method returns the single object that maximizes (or minimizes, as selected by the second argument) the provided function. As an example, the following call finds the object closest to the agent:

```
Object *nearest = memory->find(DistanceObjectFn(self),
                               Memory::FUNCTION_MINIMIZE);
```

This example is not very useful since the nearest object to the agent is, of course, the agent itself. Another more complex example finds the nearest building to the agent:

```
Object *nearest = memory->find(AddObjectFn(DistanceObjectFn(self),
                                           ClassObjectFn<Building>()),
                               Memory::FUNCTION_MINIMIZE);
```

See Section 5.2.2 for another example, which uses an agent defined object function.

The `find_list` method will return an array of all the `Objects` whose function value is larger (or smaller) than the specified threshold. The returned array is `NULL` terminated. The array is also newly allocated and it is the caller's responsibility to free it, with "`delete [] return_value;`".

4.2.2 Finding Paths

These methods are useful for finding paths, which are lists of objects that are neighboring according to the GIS data (e.g. roads, intersections, buildings).

Memory.hxx

```
  Object **find_path(const Location &origin,
                     const ObjectFn &destination,
                     const ObjectFn &heuristic =UniformObjectFn(),
                     const ObjectPairFn &edge_cost =TraversalObjectPairFn());

  bool verify_path(Object **path,
                   const ObjectPairFn &edge_cost =TraversalObjectPairFn());

  Object **update_path(Object **path, Object *here,
                       const ObjectFn &destination,
                       const ObjectPairFn &edge_cost =TraversalObjectPairFn());
```

The `find_path` method is a completely general search algorithm for finding paths. It takes an origin to begin the search. It then takes two object functions and an object pair function (only one is mandatory) that guide the search. The mandatory argument is an object function that determines the destination. The domain of this function is the goal states for the search. The second object function is a heuristic function that estimates the distance from a particular state to the goal. Finally an object pair function sets the cost associated with traversing from one state to another. The default for the heuristic function is the uniform function, and for the cost is the `TraversalObjectFn`. It returns a `NULL` terminated array of neighboring objects that define the path. As with `find_list` it is the responsiblity of the caller to free this array with "`delete [] return_value;`".

The method implements the A* search algorithm, therefore if an admissible heuristic is provided it will return a shortest (least cost) path. For example the following call returns the shortest path from object `A` to object `B` using euclidean distance as an admissible heuristic:

```
memory->find_path(Location(A), ObjectObjectFn(B), DistanceObjectFn(B));
```

The `verify_path` method takes a `NULL` terminated array and verifies that the path is still valid, i.e. each object is a neighbor of the previous object and each is in the domain of the cost object function.

The `update_path` method takes a `NULL` terminated array, a current location, and a destination object function. It then tries to find a valid subpath from the current location to a valid destination. This is useful for following a path, since it does not have to incur the computational cost of a new call to `find_path`. If there is no valid subpath or the path parameter is `NULL` then it frees the path and a `NULL` pointer is returned. This method is destructive, i.e. it modifies the path array directly. See Section 5.2.2 for an example of how to use this method.

Memory.hxx

```
  Object *path_destination(Object **path);
  Object **path_subpath(Object **path, Object *origin);
  double path_cost(Object **path,
                   const ObjectPairFn &edge_cost =TraversalObjectPairFn());
```

These methods provide useful information about paths. `path_destination` returns the last object in a given path. `path_subpath` returns a pointer into the path array that corresponds to the subpath that starts from the given origin. `path_cost` returns the sum of the edge costs along a path. With the default `edge_cost` argument this returns the length of the path.

5 Example

This section presents a comprehensive example of using the ADK to build a simple fire fighting agent. The agent searches for shortest paths to nearby fires and then attempts to extinguish them. It also sends messages to other agents with information about the status of roads, and incorporates these same messages from other agents in its view of the world.

The complete source code for this example can be found in the `examples` directory of the ADK distribution. The directory also contains a simpler example: a sample civilian that acts similar to the civilian distributed with the RoboCup Rescue distribution, but which makes use of the ADK.

5.1 Main

The `main()` function for our example is designed to make use of the feature of having multiple agents share the same communication port. It is also general and could be used for any agent derived from the `Controller` class.

main.cxx

```
  Controller::setup_connection(host, port);

  while(number != 0) {
    Controller *a = new CONTROLLERCLASS();
    if (!a->connect()) break;
    number--;
  }

  Controller::go();
```

First the connection to the kernel is established with the call to `setup_connection`. The `while` loop will then connect at most `number` agents of the type `CONTROLLERCLASS`. For this specific example the class would be `FireAgent`. After the agents are connected, the agents are started by calling `go()`.

5.2 A Fire Agent

This simple fire fighting agent illustrates a number of the features of the ADK, both the `Controller` and `Memory` classes. The key features of the implementation are described here.

5.2.1 Defining an Agent

An agent is defined as a class derived from `Controller`. The agent in this example overrides the `act` method, of course, but also overrides the `sense_object` and `hear` methods to implement the communication of changes in the environment to other agents.

FireAgent.hxx

```
using namespace Rescue;

class FireAgent : public Controller
{
protected:
  Memory *m_memory;
  Object **m_path;

  virtual void act();
  virtual void sense_object(S32 time, Object *o, Input &input);
  virtual void hear(Object *sender, const char *message);

public:
  FireAgent(int type = (1 << AGENTTYPE_FIRE_COMPANY));
};
```

5.2.2 The act Method

An agent's behavior is mostly defined in the act method, which is responsible for examining the state and then making calls to various act_* methods to perform primitive actions. In this example, the behavior of the agent is put out any fire in the current location, otherwise find the nearest fire and move there.

The implementation defines a special object function, TargetBuildingObjectFn, which is derived from the DistanceObjectFn, but restricts the domain to only burning buildings. This can then be used to find the nearest burning building.

FireAgent.cxx

```
struct TargetBuildingObjectFn : public Memory::DistanceObjectFn {
  virtual bool consider(const Object *o) const {
    return (o->type() == TYPE_BUILDING &&
            ((Building *) o)->fieryness() >= 1 &&
            ((Building *) o)->fieryness() <= 3);
  }
  TargetBuildingObjectFn(const Object *o) : DistanceObjectFn(o) {}
};
```

The act method begins by checking if the current location is a burning building and if so extinguishing it.

FireAgent.cxx

```
void FireAgent::act()
{
  MotionlessObject *here =
    dynamic_cast<MotionlessObject*>(m_self->position());

  // Put out fire in current location.
  if (here->type() == TYPE_BUILDING) {
    Building *b = (Building *) here;

    if (b->fieryness() >= 1 && b->fieryness() <= 3) {
      act_extinguish_simple(b);
      return;
    }
  }
```

If the current location is not burning it needs to find a path to a burning building. In order to help aid the search we use a heuristic function which is the Euclidean distance to the nearest building (not necessarily the easiest or fastest to reach). This is not an admissible heuristic for finding the shortest path to any burning building, but this is not a big concern in this example.

Since searching for paths can be computationally intensive, we use the method update_path to try and reuse a path as long as it remains valid.

FireAgent.cxx

```
  // Otherwise, go toward the nearest fire.
  // Update any existing path.  Otherwise generate a new path.
  m_path = m_memory->update_path(m_path, here, TargetBuildingObjectFn(here));
```

If the agent has no path to begin with or there's no valid subpath due to either blocked roads or the building is no longer burning, then the agent must generate a new path.

FireAgent.cxx

```
  if (!m_path) {
    Object *target = (Building *) m_memory->find(TargetBuildingObjectFn(here),
                                                 Memory::FUNCTION_MINIMIZE);
    if (!target) return;

    m_path = m_memory->find_path(Memory::Location(m_self),
                                 TargetBuildingObjectFn(here),
                                 Memory::DistanceObjectFn(target));
  }
```

The path is generated by first finding the nearest burning building by the call to find. This is then used to create a heuristic function for the call to find_path. The goal of the search function is an Object in the domain of the TargetBuildingSearchFn, i.e. a burning building.

Since it is now either using its old path or has generated a new one, it can simply call `act_move` to follow the path.

FireAgent.cxx
```
  // Follow the path.
  if (m_path) act_move(m_path);
}
```

5.2.3 The `sense_object` Method

The `Controller` class handles updating an agent's view of the world when new sensations are received. Agents though may want to do additional processing of state changes. This can be done by overriding the `sense_change` method or the `sense_object` method. In our example the fire agent will send a message to the other agents when a road that was thought open is discovered to be blocked, or vice versa. Since this requires the agent to examine both the object before and after the object is updated the `sense_object` method is overriden.

FireAgent.cxx
```
void FireAgent::sense_object(S32 time, Object *o, Input &input)
{
  switch(o->type()) {
  case TYPE_ROAD: {
    Road *r = (Road *) o;

    int was_clear_h = Memory::TraversalObjectPairFn().consider(r, r->tail());
    int was_clear_t = Memory::TraversalObjectPairFn().consider(r, r->head());

    Controller::sense_object(time, o, input);

    int is_clear_h = Memory::TraversalObjectPairFn().consider(r, r->tail());
    int is_clear_t = Memory::TraversalObjectPairFn().consider(r, r->head());

    if ((was_clear_h ^ is_clear_h) || (was_clear_t ^ is_clear_t)) {

      char msg[256];
      sprintf(msg, "roadinfo %ld %ld", r->id(), r->block());

      Object **other_agents = m_memory->find_list(Memory::IsAgentObjectFn());
      act_tell_list(other_agents, msg);
      delete [] other_agents;
    }

  } break;

  default:
    Controller::sense_object(time, o, input);
  }
}
```

The `sense_object` method gets called for each object that the kernel sends sensing information. In this example we are interested in examining road objects, so for other objects we just call the parent method. For roads we first store wether they are currently blocked (since a road may be blocked in only one direction we must store both.) We then call the parent method, "`Controller::sense_object(time, o, input);`", and afterwards check if the road's status has changed. If it has changed it uses `find_list` to retrieve a list of all the agents and then uses the action `act_tell_list` to pass the message to the other agents. It is very important that all calls to a derived `sense_object` call `Controller::sense_object` or the object will not actually be updated.

5.2.4 The `hear` Method

In the previous section the agent used the `act_tell_list` method to communicate to the other agents. The `hear` method can be defined in order to process communications from other agents. In this example, the fire agent needs to update its state of the world based on other agents' messages about the status of roads. This is done rather simply by parsing the message and updating the memory's stored object.

FireAgent.cxx

```
void FireAgent::hear(Object *sender, const char *message)
{
  const char *complete_message = message;
  char type[32];

  sscanf(message, "%s", type);
  message = strpbrk(message, " \t");

  if (strcmp(type, "roadinfo") == 0) {
    Id id;
    char status;

    sscanf(message, "%ld %c", &id, &status);
    if (Road *r = lookup<Road>(id)) {
      r->setBlock(m_time, (status == 'B') ? r->width() : 0);
    }
  } else {
    fprintf(stderr, "[%ld] Cannot understand message, \"%s\".\n",
            m_id, complete_message);
  }

}
```

6 Changes

Although, this is the first version it is hoped that the API will remain fairly stable so that agent development can begin quickly. Of course future versions of the RoboCup Rescue Simulator may require more encompassing changes to the ADK interface. In the meantime, updates will primarily focus on additional "tools" for the "tool box", both more complex agent actions and more useful tools for retrieving information about the environment. Recommendations of these tools or improvements to the ADK are welcome to be sent to the author or discussed on the RoboCup Rescue mailing list.

This document will attempt to maintain a list of the changes to the existing API that may require changes to agents built using a previous version of the ADK. It is hoped that these will be kept to a minimum.

6.1 Changes From Version 0.2

- Renamed `ScoreFn` and derived classes to `ObjectFn`. Changed the `find_path` method to take an `ObjectPairFn` as its edge cost function, so that roads blocked in only one direction could be handled.
- `sense_object` replaces what `sense_change` attempted, and the arguments for `sense_change` were simplified.