1.0 Introduction

The Ateams Toolkit provides a way to rapidly and flexibly build and dynamically reconfigure Ateams. This user's guide gives an overview of the toolkit's capabilities. The encapsulation of code existing in non-toolkit form is also illustrated through the example of the Travelling Salesman's Problem (TSP). This guide is not intended for naive users entirely unfamiliar with Ateams, Unix, PVM, or C++. This document is a draft, and we would appreciate any comments that would increase its clarity, usefulness, and correctness. Our e-mail addresses are: pchang@cs.cmu.edu (Philip Chang), jmd@cs.cmu.edu (John Dolan), and mt1p@edrc.cmu.edu (Michael Terk).

2.0 Running the Toolkit

Before running the toolkit, there are several things that you must set up:

- Set all the environment variables that will be used in the programs. "PVM_EXPORT" all these variables so that programs on others hosts can also recognize them. One of the variables that must be set regardless of application is "AGENT_FILE" which tells the toolkit where the agent and memory description files are located and what their names are.
- Clear any previous PVM processes or demon files.
- Start the PVM demon.
- Start the toolkit with the command: "ateam SITEDEFAULTS.tcl".

A script file which executes all the above commands automatically can be found in Appendix L.

Once the toolkit is started, a previously stored Ateam configuration can be loaded or a new Ateam can be created. These operations should be self-evident from the iconic interface and should not require further explanation.

There are, however, a few "hidden" keystrokes that are used to dynamically reconfigure the Ateams:

- Double-clicking the left mouse button an agent or memory icon pops up a window showing the description file of the agent or memory.
- Single-clicking the middle mouse button on an icon allows one to select the machine name on which the agent or memory should be running. The toolkit otherwise assigns agents and memories to machines on a round-robin basis.
- Single-clicking the right mouse button on an icon allows one to set the parameters of an agent. This can be done both during runtime and before or after a run.
- Simultaneously depressing the Shift key and single-clicking the left mouse button on an agent icon brings up a small menu which allows one to start, stop, pause, or resume that agent.

If you discover a bug or have suggestions for improvements while operating the toolkit, click on the bug report button to give us a written report.

3.0 Building an Ateam with Existing Agents and Memories

The toolkit expects a collection of agents and memories that have been encapsulated in an "toolkit-friendly" way. The expectation is that you bring to the toolkit algorithms coded in C++ (i.e., "operators") which correspond to various ways of solving or partially solving the problem you are interested in. The process of encapsulation includes defining a common solution structure for all operators, including certain header files which access system definitions and resources, and defining functions for reading, writing, and otherwise handling your particular solution structure. Encapsulation is exemplified in the tutorial in Section 4 below.

Given such an encapsulated collection of agents and memories, it is additionally necessary to create description files for the agents and memories so that the toolkit understands their types and the required parameters. The creation of description files is also described in the tutorial.

Once the description files have been created, run the toolkit and simply use the Add Agent, Add Memory, and various connection buttons to create and configure an Ateam.

4.0 Tutorial: How to build an Ateam with your own data structure and agents

This is a tutorial demonstrating how one can build an Ateam to fit within the framework of the toolkit. It is assumed that one already has the code for the operators or algorithms. The process can be divided up into three major parts:

- Construction of the memories
- Construction of the agents
- Creation of the description files for the agents and memories.

The easiest way to learn the process of creating the Ateam is to follow an example. The examples are mainly in C++ and a certain proficiency in the language is assumed.

4.1 Construction of a memory

The construction of a memory involves creating two files: a header file and a C++ file.

4.1.1 Solution data header file

The header file contains the declaration of the solution data structure. The data structure must be derived from the pre-defined ATeamData class and therefore must include the following includes and virtual function declarations:

```
#include "ATeamData.h"
#include "Channel.h"
ReadATeamData(Channel &channel);
int WriteATeamData(Channel &channel) const;
```

Here is a general template for a header file:

```
#include <iostream.h>
#include <fstream.h>
#include <assert.h>
#include "ATeamData.h"
#include "Channel.h"

class Solution: public ATeamData
{
  private:
        // Solution data variable definitions and private initialization functions
  public:
        // Solution data public initialization and destruction functions

  // Virtual functions
  ReadATeamData(Channel &channel);
   int WriteATeamData(Channel &channel) const;

  // Methods accessing solution data variables
}
```

Here is a specific example of a header file (for the TSP):

```
#include <iostream.h>
#include <fstream.h>
#include <assert.h>
#include "ATeamData.h"
#include "Channel.h"

class PathArray: public ATeamData
{
  private:
        int num_cities;              // Number of cities (length of PA array)
        int *PA;                     // PathArray array (set of city indices)
        void InitSpace(int rows);    // Function call to create a new PathArray
  public:
        PathArray() {num_cities=0; PA=0;}; // Various PA initialization functions
        PathArray(Channel &channel);
        PathArray(int rows);
        ~PathArray();

  // Virtual functions
  ReadATeamData(Channel &channel);
   int WriteATeamData(Channel &channel) const;

  int GetEntry(int row);
  void SetEntry(int row, int value);
  int GetNumberCities() {return num_cities;};
  void SetNumberCities(int value);
  // Conversion
  int const operator[](int pos);
}
```

In this example, the num_cities and PA variables in the private section of the definition are the solution data, and the methods GetEntry, SetEntry, GetNumberCities, and SetNumberCities allow one to read and write the solution data.

4.1.2 Solution data C file

In the C file, the ReadATeamData() and WriteATeamData() functions have to be defined. They read and write from a variable of the generalized type "Channel" into the data structure using the following pre-defined set of functions. The functions must be defined so that the data in the data structure are read in one at a time according to their types, e.g. integer, float, vector of integers, etc.

```
virtual int ReadInt(int &i)=0;
virtual int ReadFloat(float &f)=0;
virtual int ReadDouble(double &d)=0;
virtual int ReadIntVector(int *i, int len, int st=1)=0;
virtual int ReadFloatVector(float *f, int len, int st=1)=0;
virtual int ReadDoubleVector(double *d, int len, int st=1)=0;
virtual int ReadString(char *&str)=0;

virtual int WriteInt(int i)=0;
virtual int WriteFloat(float f)=0;
virtual int WriteDouble(double d)=0;
virtual int WriteIntVector(int *i, int len, int st=1)=0;
virtual int WriteFloatVector(float *f, int len, int st=1)=0;
virtual int WriteDoubleVector(double *d, int len, int st=1)=0;

virtual int WriteString(char *str)=0;
```

Additionally, any other methods declared in the solution header file must be defined in the C file.

4.2 Construction of an agent

Since an agent is composed of four parts, operator, scheduler, searcher and selector, constructing an agent involves the construction and assembly of the four parts.

4.2.1 Construction of an operator

An algorithm already coded as a C procedure can be converted into a toolkit-compatible operator by changing its declaration to the following form:

int NameOfOperator(Agent &agent, ATeamData &data);

Appendices A through E give examples of various operators.

4.2.2 Construction of a scheduler

The only type of scheduling implemented in the toolkit allows specification of the amount of time to sleep. This is done either by pre-specifying pause times for each agent, or by specifying the pause times through the user interface dynamically. The second mechanism can be done by using "agent.GetValue("variable_name")" to get the value of the variable, and use "agent.Link("variable_name", pause)" to link the variable to the internal variable "pause", so that whenever the value of "variable_name" changes, the pause period will be changed automatically.

The following is an example of a scheduler:

```
#include "agent.h"
#include "ateam_def.h"

extern "C" {
extern int usleep(int);
};

static double pause = 0.0;

static int SetUpLink(Agent &agent)
{
 char *var;

 if (var = agent.GetValue("agent_pause")) {
 sscanf(var, "%lf", &pause);
 }
 agent.Link("agent_pause", pause);
}

double SchedulerWithPause(Agent &agent)
{
 static int first = 1;

 if (first) {
 SetUpLink(agent);
 first = 0;
 }
 return pause*1000000;
}
```

As illustrated above, the declaration of a scheduler must be of the form:

double SchedulerWithPause(Agent &agent);

4.2.3 Construction of a searcher

Since the actual searching of solutions takes place in the memory, the only job the searcher needs to do is to return the desired search string. The supported unit search strings are listed in section 2.2.3.2.

The following is an example of a searcher:

```
class Agent;

char *NewLatenessNotFull90Searcher(Agent &agent)
{
 static char str[] = "count < 90 & set Lateness & new";        // Specifying the search
 return str;
}
```

As illustrated above, the declaration of a searcher must be of the form:

char *NameOfSearcher(Agent &agent);

4.2.4 Construction of a selector

The implementation of a selector involves reading in a list of solutions from "solution_list" and returns the solution ID of the one that is the most suitable based on certain criteria.

The declaration of the selector must be of the form:

static int NameOfSelector(Agent &agent, list<int> &solution_list);

The following is an example of a selector:

```
#include “LEDA/list.h”
#include “ateam_def.h”

class Agent;

extern “C” {
extern long random(void);
};


int RandomSelector(Agent &agent, list<int> &solution_list)
{
 int place;

 if (solution_list.empty())
 return ERROR_SOLUTION_ID;

 place = random()%solution_list.size();
 return solution_list[solution_list[place]];
}
```

4.2.5 Agent formation

The operator, scheduler, searcher, and selector are compiled together to form an agent on the basis of commands in the makefile.

4.3 Creation of the description files

In order for the GUI to include a memory or agent in its displayed lists of available memories and agents, a description file must be created. The description file gives information about permissible connections with other memories or agents, as well as other information detailed below.

The creation of description files involves creating two types of files: file_list.dat and agent_description_file.des.

The file_list.dat file includes the following information: the path of the agent and memory description files and the names of the description files.

The format is as follows: First the path of the description files, then a list of agent description file names, one on each line. At the end of the file names, a `-' is needed to separate the agent description files from the memory description file names.

Each agent has its own agent_description_file.des. This description file contain the following information:

- NAME: This is an arbitrary name for the agent or memory, preferably related to its function. This is the name that will appear in the GUI listing of agents of memories.
- TYPE: For a memory, TYPE must be one of the memory types defined in the memory process. In the line planner, for example, the only memory type defined is PM (Priority Matrix). For an agent, TYPE must be one of the following currently defined agent types: seeder, destroyer, modifier, eval, or monitor.
- DESCRIPTION: This is free text describing what the agent or memory does.
- END_DESCRIPTION
- READ_DATA_TYPES: The data types that the agent requires for input.
- WRITE_DATA_TYPES: The data types the agent requires for output.
- SWITCHES: Can be one of the following: iterations, evals, ateam, name, or cmd. The use of these switches will be explained below.
- VARIABLES: These are the variables used in the code to specify the various changing parameter such as the file name to write the results or the length of time a scheduler should pause. VARIABLES can be changed dynamically.
- ARCHITECTURES: This field specifies the type of computer this agent is able to run on. For example, this field would take a value the value SUN4 when the code is compile for Sun SPARCstations.
- MACHINE: This is to specify where the agent resides, it can take two types of values: AFS or machine name. If AFS was specified, the agent resides on AFS and thus can be executed from any machine of ARCHITECTURES types. If, however, the name of a machine is specified, that indicates the agent residing on one of machines and thus can only be executed on that machine.
- PATHNAME: The path to the agent executable.
- COMPOSITION: The names of the various components that makes the agent. Namely, these components are: the scheduler, the searcher, the selector and the operator.

Each memory also has a memory_description_file, which is very similar to an agent description file, except that it does not have the READ_DATA_TYPES, WRITE_DATA_TYPES, and COMPOSITION fields, since it does not read or write any data and it does not have the four components that agents do. However, it does require one piece of information:

- EVAL_TYPE (for memories only): This is a list of names of the evaluation fields defined for solutions contained by this memory. The line planner memory, for example, uses only "Lateness". If additional evaluation fields were defined, these would be separated by spaces, e.g. "Lateness Fieldname2 Fieldname3...".

Appendices F and G give examples of a file_list.dat and an agent_description_file.des for the TSP. [A memory_description_file.dat should be added.]

4.4 Start-up

Appendix L gives a sample toolkit start-up script for the TSP.

5.0 General Notes on Encapsulation

While most of the variables users have to deal with are of user-defined types, there are a few toolkit-defined variables that the user may need to be aware of. Appendices H through H are the three class definitions (Agent, MHistory, and ATeamData) that one will mostly come in contact with. A set of free data structure and algorithms called "Library of Efficient Data Types and Algorithms (LEDA)" was used extensively. We will not include any LEDA data types or algorithms in this user guide; however, if you need more information. please refer to "The LEDA User Manual, Version 3.1, Stefan Naher, Max-Planck Institut fur Informatik, 66123 Saarbrucken, Germany".

5.1 Basic Constructs of the Memory

A memory is a group of solutions stored together. A solution has three parts: data, evaluation, and history. The evaluation and history structures are predefined: the evaluation is an array of floats, and the history is a set of statistics on the progress of the memory. The data portion, however, must be defined by you during the encapsulation process. To construct a memory, you must define a data structure appropriate for the problem and the helper functions for access to and manipulation of the data structure.

5.2 Basic Constructs of the Agents

An agent is composed of four parts: the operator, the scheduler, the searcher and the selector. Each is described separately below.

5.2.1 Operators

The operator is the central part of an agent, it decides what is to be done to the solution data or evaluation. It is in most cases the component within an agent that differentiates between the agent types. There are six types of agent currently supported by the toolkit: Modifiers, evaluators, destroyers, seeders, monitors and transfer agents. Each functions differently and has different requirements for the information needed and the information provided. They will be discussed in more detail in the subsections below. The six types of agents can be classified into two categories: constructor agents and destructor agents.

For all agents, the generic header files ATeamData.h and agent.h allow the agent to properly interact with memory.

5.2.1.1 Constructor Agents

A constructor agent is one that reads a solution and either makes modifications according to its abilities or makes no changes to the solution pool. None of the constructors will cause the number of solutions in the memories to decrease.

1. Modifier Agent

A modifier agent takes a solution from memory, modifies it and insert it back into memory.

"ATeamData" is a virtual class. Objects of this data type are passed around to agents and memories. To use the information contained in ATeamData, simply cast it to the data type that you have defined for the memory.

2. Evaluator Agent

The evaluator takes the data in the form of ATeamData as in the modifier, evaluates it and returns a single float type value back to memory. Since a solution can be evaluated based on several criteria, it is therefore crucial to assign each evaluation a name, this is done by assigning the variable "e_name" the name of the evaluation, as shown in the example in Appendix C.

3. Seeder Agent

A seeder creates one solution and returns a pointer to this newly created solution. The example in Appendix A shows that the new solution is created by calling the creator function.

4. Monitor Agent

There are currently two types of monitor agents in use: displayer type agents and agents that monitors the memory history.

- Memory Monitor Agent

Since the memory keeps track of information such as the total number of solutions, the number of evaluated solutions, etc., a memory monitor agent's job is to either display these information or do further processing and displays the processed result. The purpose of this type of agent is to display information to help the user with the dynamic control of the Ateam, such as which agents should be started at a certain time and what is the sequencing of the agents to get better performance.

Currently there is an agent called TkMemMonitor which is available for the display of memory statistics and the plotting of the best, average and worst solutions in memory. This agent is generic for all applications.

An example of a memory monitor agent can be found in Appendix D.

- Displayer Agent

A displayer agent is used to display graphical information about the operation of an Ateam. In the TSP, for example, a displayer agent is used to plot the city nodes and a tour connecting them or some subset of them.

An example of a displayer agent can be found in Appendix E.

5. Transfer Agent

Transfer agents are also agents without an operator. The basic operation of a transfer agent is to send a solution from one memory to another without making any modifications unlike a modifier agent. The solution which is to be sent is determined by the searcher and the selector.

5.2.1.2 Destruction Agent

A destruction agent is the counterpart of the construction agent. It runs to preserve the size of the population. The destruction agents plays as important a role as the destructor in eliminating solution that it thinks are unlikely to yield good solutions.

1. Destroyer Agent

Currently, the destroyer is one of the agents that has no operator. The basic operation of a destroyer is sending a signal to the memory telling it to destroy the solution returned from the searcher and selector.

5.2.2 Searcher

A searcher sends a request to the memory telling it the type of solution it wants. Currently the set of supported search queries include:

- *: Random solution.
- count: The number of solutions in memory. For example: count > 70.
- new: The solution which the agent has not seen before.
- max: The solution with the maximum evaluation. For example: max TourLength.
- min: The solution with the minimum evaluation. For example: min TourLength.
- setcount: The number of solutions which the evaluation has been set. For example: setcount TourLength > 50.
- set: The solutions which have their evaluation set. For example: set TourLength.

- unset: The solutions which do not have their evaluation set. For example: unset TourLength.

A list of possible search strings can be found in Appendix K.

5.2.3 Selector

Given the list of solution identifiers that matches the criteria specified by the searcher, the selector's tasks is to single out one solution which the operator is going to work on.

5.2.4 Scheduler

A scheduler decides when an agent should run. Currently, this is implemented by deciding how many seconds to pause in between executions.

6.0 Memory

A memory is a process that stores and retrieves solutions generated by agents in the Ateam and the evaluations associated with these solutions.

At this time we have developed two sample implementation of memory processes. The memory processes are designed to store solution data for the line-planner problem using the priority matrix representation and another to store cities in a tour for the TSP application. In addition, for each solution, the memory process creates and manages a vector of evaluations, such as cumulative lateness, assigned to the solution.

At present, the memory process supports the following external requests:

- AGENT-CHECK-IN - each agent that either reads from or writes to a memory process must check in to the memory. Each agent is then assigned a unique identifier and that information is stored in the memory process.
- AGENT-CHECK-OUT - each agent that terminates its processing must notify all memories it has accessed. The unique identifier assigned to the agent during check-in is released and can be re-assigned.
- CREATE-SOLUTION - the memory process receives a custom defined data structure and creates a solution used to store the custom defined data structure. Each solution is assigned a unique identifier. In addition, the memory creates an empty evaluation array that will be used to store the evaluations associated with the solution.
- GET-SOLUTION - the memory process receives a solution identifier, and if the solution with this identifier exists, returns the custom defined representation to the requesting agent.
- DELETE-SOLUTION - the memory process receives a solution identifier, and if the solution identifier is valid, the data portion of the solution is deleted from memory.

- SET-EVALUATION - the memory process receives an evaluation name (e.g., tour-length for the TSP application) and the value of the evaluation. Each evaluation is assigned to the unique location in the evaluation vector and the value of the evaluation is stored at that location. In addition, the identifier of the agent providing the evaluation is also recorded along with the evaluation in case this information is needed in the future.
- GET-EVALUATION - the memory process receives a list of evaluation names. Each name is converted to the location in the evaluation array and the values of evaluations are returned to requesting agent.
- SEARCH - the memory process receives a search string. The search string is interpreted by the memory process and the identifiers of solutions that match the specifications in the search string are returned to the requesting agent. The current implementation of the memory process supports search strings containing boolean expressions involving the number of solutions in the memory and the value of a specific evaluation field. It also supports queries relating to the history of solution and a random selector.

Appendix A: TSP Example: Seeder

```
#include “PathArray.h”
#include “TSP_defines.h”

class Agent;

PathArray *Seeder(Agent &agent)
{
 PathArray *data1 = new PathArray(atoi(getenv(“CITIES”)));
return (data1);
}
```

Appendix B: TSP Example: Modifier

```
#include “PathArray.h”
#include <fstream.h>
#include <iostream.h>
#include “ateam_def.h”
#include “ATeamData.h”
#include “eval.h”

extern “C” {
 extern long int random();
}

class Agent;

int AI(Agent &agent, ATeamData &data1)
{
PathArray *tour_ptORIG = (PathArray *) &data1;
// AI operator inserted here
return ATEAM_OK;
}
```

Appendix C: TSP Example: Evaluator

```
#include <stdio.h>
#include <stdlib.h>
#include “TSP_defines.h”
#include “PathArray.h”
#include “ateam_def.h”
#include “eval.h”
#include “util.h”

class Agent;

int Evaluate(Agent &agent, ATeamData &a_data, char **eval_name, float *value)
{
 static char e_name[]=”TourLength”;

 PathArray *data = (PathArray *)&a_data;
// Evaluate here
 return ATEAM_OK;
}
```

Appendix D: TSP Example: Memory Monitor

```
#include "history.h"
#include "agent.h"
#include "ateam_def.h"
#include "counter_def.h"
#define DEFAULT_FILE "memory.status";

class Agent;
static int counter = 0;
char *status_file;

int PrintMemStatistics(Agent &agent, MHistory &history)
{
 if (!counter) {
 if (!(status_file = agent.GetValue("status_file"))) {
 status_file = DEFAULT_FILE;
 }
counter++;
 }
 fstream outStream(status_file, ios::app);
 if (!outStream) {
 cerr << "Unable to open " << status_file << " for output";
 return 0;
 }

 outStream << "After " << history.GetCounter (ATEAM_INSERT_SOLUTION)
 << " iterations :" << endl;
 cerr << "After " << history.GetCounter (ATEAM_INSERT_SOLUTION)
 << " iterations :" << endl;

 for (int i = 0; i < history.GetEvaluationCount(); i++) {
 outStream << "Evaluation" << i << ": (" << history.GetMaxEval(i)
 << ", " << history.GetAvEval(i) << ", "
 << history.GetMinEval(i) << ")" << endl;
 cerr << "Evaluation" << i << ": (" << history.GetMaxEval(i)
 << ", " << history.GetAvEval(i) << ", "
 << history.GetMinEval(i) << ")" << endl;
 }
 for (i = 0; i < history.GetCounterLength(); i++)
 cerr << CODE_LABELS[i] << " is: " << history.GetCounter(i) << endl;
 outStream.close();
}
```

Appendix E: TSP Example: Display

```
#include "ATeamData.h"
#include "agent.h"
#include "PathArray.h"

class Agent;
static int flag = 0;

int DisplayTour(Agent &agent, ATeamData &a_data)
{
 PathArray *data = (PathArray *)&a_data;

 if (!flag) {
 // First time through, perform initialization and setup
 // Set flag to ensure this section is only executed the first time
 flag = 1;
 }

 // Extract, manipulate if necessary, and display the relevant
 // fields in the structure *data (solution data).
}
```

Appendix F: TSP Example: Descriptor File

```
/afs/cs.cmu.edu/project/edrc-22/project/ateams/tk_tcl/gui/dataTSP
evaluate.des
seeder.des
destroyer.des
killbest.des
intersection.des
ai.des
two_opt.des
memmonitor.des
tkmemmonitor.des
ilk.des
lk.des
or.des
tourdisplay.des
bab.des
kick.des
lk_incomplete.des
afi.des
-
mem.des
```

Appendix G: TSP Example: Description File

```
NAME Intersection

TYPE modifier

DESCRIPTION This is the intersection agent
END_DESCRIPTION

READ_DATA_TYPES PathArray TourLength

WRITE_DATA_TYPES PathArray


SWITCHES {iterations 0}

VARIABLES {agent_pause 0}

ARCHITECTURES SUN4

MACHINE AFS

PATHNAME /afs/cs.cmu.edu/project/edrc-22/project/ateams/TSP/bin/Intersection

COMPOSITION operator4 searcher3 selector1 scheduler1
```

Appendix H: Agent Class

```
class Agent: virtual public TclPvmInterp {
private:
char *name;
PvmChannel my_tid; // the pvm task id for the agent
PvmChannel ateam_tid; // the pvm task id for the ateam running agent
char *ateam_name;
void InitSpace(const char *name, const char *ateam_name); CheckIntoMemory(Chan-
nel &memory_tid, int code);
CheckOutOfMemory(Channel &memory_tid, int code);
public:
Agent(const char *name, const char *ateam_name);
virtual ~Agent();
inline int CheckInRead(Channel &memory_tid); //returns agent id
inline int CheckInWrite(Channel &memory_tid);
inline int CheckInDestroy(Channel &memory_tid);
int CheckInToATeam(char *ateam_name, PvmChannel &ateam_tid);
inline int CheckOutRead(Channel &memory_tid); //return ATEAM_OK/ERROR
inline int CheckOutWrite(Channel &memory_tid);
inline int CheckOutDestroy(Channel &memory_tid);
int CheckOutOfATeam(void);
double ScheduleAgent(double (*scheduler)(Agent &agent));
int SearchForSolutions(Channel &memory_tid, char *(*searcher)(Agent &),
 list <int>&solution_List);
int SelectSolution(int (*selector)(Agent &, list<int> &solution_list),
 list<int> &solution_list);
ATeamData *RequestSolution(Channel &memory_tid, int solution_id);
int InsertSolution(Channel &memory_tid, const ATeamData &data);
int DestroySolution(Channel &memory_tid, int solution_num);
int EvaluateSolution(int (*evaluator)(Agent &, ATeamData &data, char **name,
 float *value), const ATeamData &data, char **eval_name, float *eval_value);
Evaluation *RequestEvaluation(Channel &memory_tid, int solution_id,
 char *eval_names);
int SetEvaluation(Channel &memory_tid, int solution_id, char *eval_index,
 float eval_value);
MHistory *RequestMemoryHistory(Channel &memory_tid, char *eval_names);
const char *Name() {return name;};
const char *ATeamName() {return ateam_name;};
virtual int Execute(void) {};
virtual double Schedule(void) {};
virtual Evaluation *RequestEvaluation(int solution_id, char *names) {
 return NULL;};
 ProcessMessages(int block=0);}
```

Appendix I: MHistory Class

```
class MHistory {
 friend Memory;
 private:
// what should be the number of counters???
 int counter_length;
 int evaluation_count;
 int *counters;
 float *max_eval;
 float *min_eval;
 float *av_eval;
 float *totals;
 int *active_totals;
 int *max_solutions;
 int *min_solutions;

 public:
 MHistory (int counter_length, int evaluation_count);
 MHistory (MHistory *copy, list<int> slots, int counter_count);
 MHistory (Channel &channel);
 ~MHistory ();
 IncrCounter (int counter_id);
 DeleteEvaluation (Evaluation *evaluation, int solution_id,
 int evaluation_count);
 AddEvaluation (float eval, int index, int solution_id);
 MHistory *CreateHistory (list<int> slots);
 WriteHistory (Channel &channel);
 int GetMaxSolutions(int slot) {return max_solutions[slot];};
 int GetMinSolution(int slot) {return min_solutions[slot];};
 float GetMaxEval(int slot) {return max_eval[slot];};
 float GetMinEval(int slot) {return min_eval[slot];};
 float GetAvEval(int slot) {return av_eval[slot];};
 int GetCounter (int slot) {return counters[slot];};
 int GetCounterLength () {return counter_length;};
 int GetEvaluationCount() {return evaluation_count;};
};
```

Appendix J: ATeamData Class

```
class ATeamData {
 public:
 virtual ~ATeamData() {;};
 virtual ReadATeamData(Channel &channel) = 0;
 virtual int WriteATeamData(Channel &channel) const = 0;
 void SendDataMessage(PvmChannel &channel, int msgtag) const;

 friend Channel& operator << (Channel &out, const ATeamData &data);
 friend Channel& operator >> (Channel &in, ATeamData &data);
};
```

Appendix K: Search String Examples

- Count

count < integer

count <= integer

count >= integer

count > integer

e.g.

count < 50

returns solutions if total number of active solutions is less than 50

- Range

range integer < float

range integer <= float

range integer >= float

range integer > float

e.g.

range 1 < 50.5

returns a solution out of set of solutions with evaluation in slot 1

less than 50.5

- Min and Max

min integer

max integer

e.g.

min 1

returns a solution with smallest evaluation in slot 1

- New

new

returns only solutions that the agent has not worked on these string can be combined with the "&" sign

so

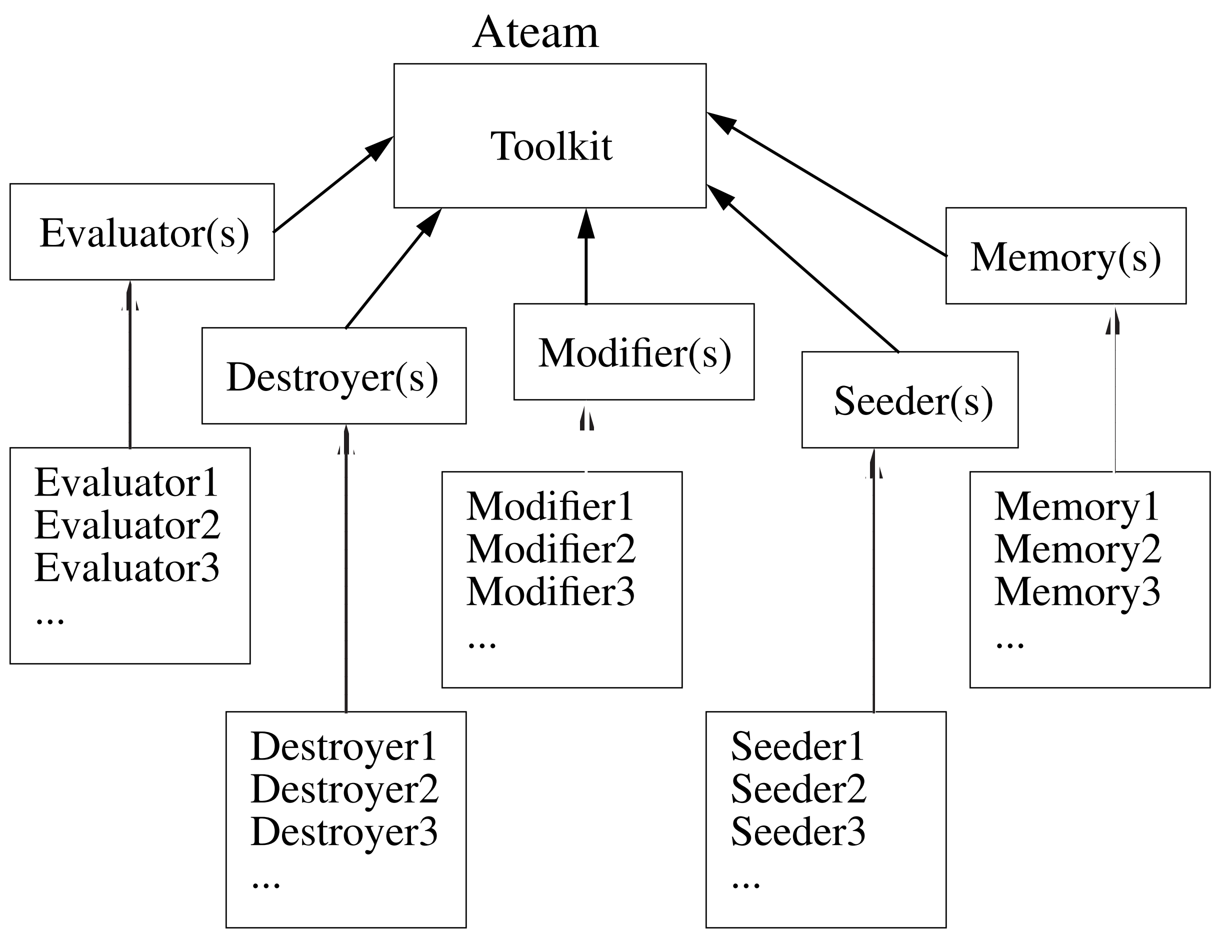
new & range 1 < 50.5

will return solutions with evalution in slot 1 less than 50.5 and that this agent has not worked on before.

Appendix L: TSP Example: Toolkit Startup Script

```
#!/bin/csh
if ($#argv) then
 setenv AGENT_FILE $1
 setenv CURRENT_DIR `pwd`
endif
if ($#argv == 4) then
 setenv DISTANCE_FILE $2
 setenv CITIES $3
 setenv COORD_FILE $4
 setenv PVM_EXPORT DISTANCE_FILE:CITIES:COORD_FILE
else
  echo "Error, proper usage is: run_tsp ateam_file distance_file total_cities
coordinate_file"
 exit
endif
if ("$DISPLAY" == ":0.0") setenv DISPLAY `hostname`:0.0
echo "Display is $DISPLAY"
xhost +rielly.edrc.cmu.edu > & /dev/null
xhost +bourne.edrc.cmu.edu > & /dev/null
xhost +bond.edrc.cmu.edu > & /dev/null
xhost +hale.edrc.cmu.edu > & /dev/null
xhost +fleming.edrc.cmu.edu > & /dev/null
xhost +fouche.edrc.cmu.edu > & /dev/null
xhost +pocono.edrc.cmu.edu > & /dev/null
/afs/andrew.cmu.edu/system/gamma/sun4_413/contributed/bin/zap -u $user -s 15
pvmd3
/afs/cs.cmu.edu/misc/pvm/sun4c_411/omega/bin/pvmd3 /afs/cs.cmu.edu/project/
edrc-22/project/ateams/hostfile.all &
/usr/local/bin/xterm -e /afs/cs.cmu.edu/project/edrc-22/project/ateams/bin/ateam /
afs/cs.cmu.edu/project/edrc-22/project/ateams/tk_tcl/gui/src/SITEDEFAULTS.tcl
/afs/andrew.cmu.edu/system/gamma/sun4_413/contributed/bin/zap -u $user -s 15
pvmd3
```

Appendix M: Toolkit Components



Appendix N: Toolkit Components