# NSF ITR Robot-PFC
# Working Memory Toolkit Tutorial

**Joshua Phillips**
**11/15/04**

## Contents

# Purpose

This document provides basic instructions for interfacing with the NSF ITR Robot-PFC Working Memory Toolkit (WMtk). The WMtk provides an extensive API that facilitates the fabrication of systems that utilize biologically-inspired working memory components. This document is not a replacement for the WMtk Specification document, which provides information on the entire WMtk API. Rather, it is provided to aid researchers in identifying which components and methods of the API are critical to fabricating a functioning working memory system and to provide guidance on their proper use.

# Introduction

The WMtk is a group of ANSI C++ clases that are designed to aid in the implementation of a biologically-inspired working memory system. In order to begin the process of developing such a system it will be important to familiarize yourself with a few key classes from from the toolkit and what the basic purpose these classes serve in the toolkit's framework. After examining the basic purpose of each of these classes, we will examine how these classes should interact to facilitate working memory.

**Chunk**

The Chunk class is basically a simple data structure that stores some arbitrary piece of information that you think might be useful for processing. This could be a coordinate, an angle, a plan, an action, or any other “chunk” of information. This class will act as a wrapper for this information and hold onto some basic data-type information in the form of a string. Here is the Chunk class's constructor and some of the most commonly used methods of the Chunk class:

```
Chunk(void* data, string type)

void* Chunk::getData()
void  Chunk::setData(void* data)

string Chunk::getType()
void   Chunk::setType(string type)
```

As you can see, the constructor takes a pointer to some memory location and a string as arguments. The pointer may point to any data structure you like and the string (a standard C++ string) allows you to store a basic amount of type information along with the pointer so that the pointer can be cast to its appropriate type later when needed. Here is an example for storing an integer value in a Chunk class:

```
int* my_data = new int;
*my_data = 5;

Chunk my_chunk((void*) my_data, “Integer”);

my_data = NULL;
```

Now `my_chunk` is storing the newly created integer for me. Later, I can retrieve the data and type information in order to make use of the data

again:

```
double* my_double_data;

if (my_chunk.getType() == “Integer”)
     my_data = (int*) my_chunk.getData();
else if (my_chunk.getType() == “Double”)
     my_double_data = (double*) my_chunk.getData();
```

It is important to note that the type information is needed to make a decision about how to cast the void* returned by getData(). You must cast the void* in order to make use of it. The Chunk class does not make a deep copy of any data structure, it only remembers the address of the data and it's type information.

**FeatureVector**

The working memory system utilizes a neural network to make decisions about memory management. Like with most neural networks, information is provided in vector form. A FeatureVector class is a wrapper for just such a vector and also provides some extra functionality for filling the vector with values. It is an important class to you, the system designer, because it will provide the interface though which you can provide Chunk and State information to the neural network so that it can make intelligent decisions about memory management.

The FeatureVector class provides many useful methods for encoding information into the class. All values are stored as doubles and (in the default toolkit configuration) can only be in the range [0,1]. Typically, you will not need to create a FeatureVector object, but you will be storing and coding values into these objects. Please see the WMtk Specification document for a full description of the FeatureVector class's methods. Later in this document, we will examine coding strategies and how to use some of this class's methods to build efficient codes for the neural network.

**WorkingMemory**

The WorkingMemory class is the workhorse of the WMtk. It provides all of the functionality thought to pertain to working memory in the system. In other words, it is a limited capacity store with the intelligence to know when it's contents should be updated and/or protected (erased and/or retained). It will make these decisions on each “time step” of the system simulation. Looking at the WMtk Specification document, it is easy to see that this is the most complex class in the toolkit. However, its operation is actually very straightforward once you know what information it requires in order to operate correctly. It will probably be one of the first objects created in your program and will

probably persist throughout the entire simulation. In most cases this will look something like the following:

```
#include <WMtk.h>
...
state_data_structure current_state;
double my_reward_function(WorkingMemory&);
void   my_state_translation_function(FeatureVector&,
                                     WorkingMemory&);
void   my_chunk_translation_function(FeatureVector&,
                                     Chunk&,
                                     WorkingMemory&);
void   my_chunk_deletion_function(Chunk&);
...
int main() {
...
int     size_of_working_memory = 4;   // Example value
int     size_of_state_vector = 20;    // Example value
int     size_of_chunk_vectors = 15;   // Example value
bool    using_actor_network = false;  // Example value
OR_CODE or_code = NOISY_OR;           // Example value
...
WorkingMemory WM(size_of_working_memory,
                 size_of_state_vector,
                 size_of_chunk_vectors,
                 &current_state,
                 my_reward_function,
                 my_state_translation_function,
                 my_chunk_translation_function,
                 my_chunk_deletion_function,
                 using_actor_network,
                 or_code);
...
// MAIN LOOP
...
}
```

The first argument to the WorkingMemory class constructor is `size_of_working_memory`. This is simply the maximum number of chunks that the working memory system can hold at any given time. The value provided is just an example of a system that can hold no more than four chunks in working memory.

I will skip the second and third arguments for now and talk about the fourth. This argument, `&current_state`, is a pointer to whatever data structure holds all information about the current state of the system and the environment in one data structure. It can be anything

since it is maintained as a void*. You can retrieve this state data structure from the WorkingMemory class later on by calling it's getStateDataStructure() method. This will be of use later on.

The second argument, `size_of_state_vector`, and the sixth argument, `my_state_translation_function`, are closely related to one another. The function is used by the working memory system to translate the current system state into a vector form that can be used by the neural network. Since the working memory system has no way of knowing how to do this, it will call your function to accomplish this goal. Inside of this function should be all of the code needed to translate the current state into vector form. Notice the function takes two arguments, a FeatureVector object and a WorkingMemory object. The WorkingMemory object will call the translation function (something you will never have to do) when it needs to translate the current state into a FeatureVector representation. It will provide a FeatureVector of the size specified by the second argument of the WorkingMemory constructor, `size_of_state_vector`, and also itself (*this) to the function. You can then use the WorkingMemory::getStateDataStructure () method inside of your function to get the state information and then translate it into the FeatureVector class.

The third and seventh arguments go hand-in-hand as well. The `my_chunk_translation_function` will work in a similar manner to the `my_state_translation_function`. However, `my_chunk_translation_function`, takes one more argument which is a Chunk to be translated into the provided FeatureVector. The WorkingMemory class is passed into the function as an argument in case the state data structure, current contents of working memory, or anything else needs to be examined to encode the Chunk information into vector form.

The fifth argument, `my_reward_function`, is similar to the others in that it will be called by the WorkingMemory class. However, it will only be called when it needs reward information about the current state. You must write this function where it takes information from the current state (accessed via WorkingMemory::getStateDataStructure()) and the contents of working memory to generate a scalar reward signal that it will then return to the WorkingMemory system. Later we will discuss reward schemes and scheduling that could potentially be used by the WorkingMemory system.

The eighth argument, `my_chunk_deletion_function`, is another function that you must implement to handle memory management issues. Once the WorkingMemory class has finished with a Chunk, it cannot properly dispose of the Chunk data on its own because it does not know how to cast the Chunk data pointer appropriately to do so. Also, you might possibly (although not likely) want to retain some or all of the old Chunks that the WorkingMemory class is finished with. Each

time the WorkingMemory class decides that it no longer needs to hold onto a Chunk, it will call this function before releasing it from the working memory store. It only takes one argument, a Chunk, and you should write this function to either delete the data pointed to by the Chunk data pointer (void*) or manage the chunk data in some other way. Again, after this function has been called on a Chunk, the WorkingMemory class will not contain this Chunk any longer, so either delete the data or store the pointer in some other manner via this function.

The ninth argument, `use_actor_network`, is currently not used by the class and will be ignored in the current version of the system. Although, it may be used in later versions. To be safe, always set it to **false**.

The tenth argument, or_code, specifies whether to add an addition coding scheme to the network. Possible values are NO_OR, MAX_OR, or NOISY_OR. NO_OR is the default, but the other values should not hurt performance. Again, to be safe about future version issues, NO_OR is probably the best option for now. This option will be discussed more later when encoding schemes are examined more in-depth.

The comment after creating the WorkingMemory object (// MAIN LOOP) specfies where the actual working memory processing will occur and is the topic of the next section.

## Processing

Now that we know how to create the WorkingMemory object that will be used by the system we need a general approach to applying this object to correctly solve some problem. Here is the basic outline of the Main Loop should look like (in pseudocode):

```
do {

     // Inialize the WM object for this episode
     WM.newEpisode(true);

     // Initialize the state data structure to match the
     // start state.
     initializeState();

     do {

          // Update the state information in the state data
          // structure to match the current state of the
          // world.
          updateState();

          // Analyze the current state of the world for
          // possible chunks to provide to the working
          // memory system.
          list<Chunk> candidate_chunks = analyzeState();

          // Provide the Working Memory system with the
          // candidate chunks for this time step.
          WM.tickEpisodeClock(candidate_chunks);

          // Choose action(s) based on working memory
          // contents and current state of the world.
          decideAction();

          if (no useful action could be discerned) {
               // Choose to perform an action that explores
               // or doesn't rely on working memory
               // contents.
               decideAutomaticAction();
          }

          // Perform action
          performChosenAction();
```

```
    } until the goal is obtained or the system fails to get
      to the goal in time, etc.


} until the desired number of episodes have been completed
```

Notice that since the state translation, chunk translation, reward, and chunk deletion functions are automatically called by the WorkingMemory object, the processing loop is rather simple. Some of the actions in the inner loop can be rearranged without really changing the processing much. This is only a draft of this document and more information will be available soon.