

Senior Honor Thesis

The implementation of DaSSF OTcl APIs

Student: Hongxia Quan

Thesis Advisor: David M. Nicol

Dartmouth Technical Report PCS-TR99-355

June 3, 1999

The implementation of DaSSF OTcl APIs

Abstract: As an extension of Tcl, Otcl provides basic functionality for object-oriented programming in scripting language Tcl. We implemented the Otcl APIs for DaSSF (a parallel simulator software written in C++ at Dartmouth College) using Tclcl software package written in University of California at Berkeley. This document discussed the issues involved in the implementation, especially the communications between C++ objects and Otcl objects required by DaSSF and the naming problems.

1.0 Introduction

The goal of S3 - scalable self-organizing simulation-- is to achieve radical improvements in speed, scalability, and manageability of modeling and simulations of very large-scale multiprotocol communications networks. The S3 consortium developed the Scalable Simulation Framework (SSF) with the goal of establishing an API for parallel simulators that support porting SSF simulation of SSF. Dartmouth SSF (DaSSF) complies with SSF API with extensions specific to the host language and its own thread system. This thesis consists of developing the OTcl APIs for DaSSF.

2.0 Background

2.0.1 Tcl

Tcl is a high-level scripting language that was developed in 1987. Tcl has several advantages used in simulation. Tcl is easy to learn and use due to its elegant simplicity and an imperative style that is immediately familiar to any programmer. Tcl is interpreted so it is highly portable and saves the time for compile. However, in large-scale simulation, Tcl also has disadvantages, the biggest being Tcl is not object-oriented so it complicates large-scale software development. Otcl is meant to overcome this shortcoming.

2.0.2 Otcl

Otcl, short for MIT Object Tcl, is an extension to Tcl/Tk for object-oriented programming. Some of OTcl's features as compared to alternatives are: designed to be dynamically extensible, like Tcl, from the ground up; builds on Tcl syntax and concepts rather than importing another language; compact yet powerful object programming system (draws on CLOS, Smalltalk, and Self); fairly portable implementation (2000 lines of C, without core hacks). A sample of Otcl program is shown in Figure 2.1.

```
Class Bagel
Bagel instproc init {args} {
eval $self next $args
$self set toasted 0
$self set flavor none
$self set bites 12
}

Bagel instproc toast {arg} {
```

```

$self set toasted [eval [$self set toasted] + 1]
}

Bagel instproc add_flavor {arg} {
$self set flavor $arg
}

Bagel instproc bite {} {
if {[[$self set bites] <= 0] } {
error "$self is eaten up"
}
$self set bites [expr [$self set bites] - 1]
}

Bagel sesame_toasted_bagel -toast -flavor sesame
Bagel onion_raw_bagel -flavor onion

```

Figure 2.1 an Otcl program which declares class Bagel and two objects of the Bagel class, sesame_toasted_bagel and onion_raw_bagel. This Bagel class has 3 member variables, toasted, flavor, and bites and three member functions, toast, flavor and bite.

With only 2000 lines code added to Tcl core, the objects in Otcl are quite limited comparing to C++. In Otcl, objects of the same class can have different member variables and even different methods. Furthermore, all member data variables and methods are public in Otcl. Nonetheless, with comparison to Tcl core, OTcl provides a simplified way to develop large-scale software using scripting language.

2.0.3 C++ with Otcl

C methods can be registered in OTcl, but Tcl core does not provide the corresponding registration with C++ object. Researchers of VINT project in UC Berkeley have developed Tccl package which enable the registering of C++ class and creation of corresponding C++ objects in OTcl. Tccl is written in C++ and provides two C++ objects: TclClass and TclObject. TclClass enables classes created in C++ side to be known to OTcl side. Suppose we have a class Bagel in C++ side (figure 2.2), through the C++ code in Figure 2.3, we will have a class Bagel in OTcl.

```

Class Bagel {
Private:
Bool toasted;
Public:
Bagel() { toasted = false; }
~Bagel();
bool toasted() { return toasted; }
void toast() { toasted = true; }
}

```

Figure 2.2 A C++ Bagel class

```

Class Otcl_Bagel : public TclClass {

```

```

Otbl_Bagel() : TclClass("Bagel");
TclObject* create(int argc, const char*const* argv) {
assert(argc >= 1);
return (new Otbl_Bagel_Object(argv[0]));
}

```

Figure 2.3 C++ code that creates class Bagel for Otbl side.

In Figure 2.3, the constructor Otbl_Bagel registers the name "Bagel" in the hashtable of TclClass in Tclcl package. When the user wrote "Bagel a_bagel_object" to declare an object a_bagel_object of Bagel class, the Otbl_Bagel::create will be called and an instance of TclObject class is created. The Otbl_Bagel_Object class can be implemented as follows:

```

Class Otbl_Bagel_Object: public TclObject, public Bagel {
public:
Otbl_Bagel_Object();
~Otbl_Bagel_Object();
int command(int argc, const char*const* argv);
};
int Otbl_Bagel_Object::command(int argc, const char*const*argv) {
Tcl *tcl;
tcl = &(Tcl::instance());
if(argc == 2)
{
if(strcmp(argv[1], "toasted") == 0)
{
tcl->resultf("%d", Bagel::toasted());
return TCL_OK;
}
if(strcmp(argv[1], "toast") == 0)
{
Bagel::toast();
return TCL_OK;
}
}
return TclObject::command(argc, argv);
}

```

Figure 2.4 an implementation of Otbl_Bagel_Object class

After "Bagel a_bagel_object" is executed, a_bagel_object is linked with a TclObject. Further widget-like invocation of a_bagel_object such as "a_bagel_object <func>" will make Tcl interpreter first searched all the methods declared on a_bagel_object's behalf on Otbl side. If such method is not found, The C++ function Otbl_Bagel_Object::command will be invoked with "<func>" as the second argument and "a_bagel_object" as the first argument. Otbl_Bagel_Object::command will recognize "<func>" either as a method for Bagel or an unknown method. Known methods are processed by calling the corresponding Bagel method and return the result to Tcl. Unknown method are returned to OTcl side with the statement "return TclObject::command(argc, argv);" and OTcl will return error to the user.

3.0 DaSSF OTcl APIs

3.0.1 The APIs

DaSSF OTcl APIs are the following:

Table 3.1 DaSSF OTcl classes and instance methods

OTcl Class	Instance Methods <argument type>	OTcl return value
Entity	now	double
	makeIndependent	double
	joinAll	no return value
	coalignedEntities	a list of Entities
	waitForever	no return value
	waitOn	no return value
	alignTo <Entity>	double
	waitOn <a list of InChannels>	no return value
	waitFor <double>	no return value
	waitUntil <double>	no return value
	startAll [<double>] <double>	no return value
	waitOnFor <a list of InChannels> <double>	integer (timeout flag)
	waitOnUntil <a list of InChannels> <double>	integer (timeout flag)
Event	save	OTcl object name of the saved Event
	release	no return value
	aliased	integer

InChannel	owner activeEvents mappedto	Entity a list of Events* a list of OutChannels
OutChannel	owner mappedto mapto <a list of InChannels or InChannel> unmap <a list of InChannels or InChannel> write <Event> [<double>]	Entity a list of InChannels double double no return value
Process	isSimple owner waitForever activeChannels waitOn < a list of InChannels> waitsOn < a list of InChannels> waitFor <double> waitUntil <double> waitOnFor < a list of InChannels> <double> waitOnUntil < a list of InChannels> <double>	integer Entity no return value a list of InChannels no return value no return value no return value no return value integer (signal timed out or not) integer (signal timed out or not)

Random**	uniform	double
	exponential <double>	double
	chisquare <long>	double
	student <long>	double
	bernoulli <double>	long
	poisson <double>	long
	permute <long>	a list of long numbers
	uniform <double> <double>	double
	erlang <long> <double>	double
	pareto <double> <double>	double
	normal <double> <double>	double
	lognormal <double> <double>	double
	equilikely <long> <long>	long
	binomial <long> <double>	long
	pascal <long> <double>	long

* The implementation of OTcl API supports a list of events returning from method activeEvents of InChannel class. However, based on the implementation of DaSSF, at most one event is returned. Therefore, the length of the list is at most 1.

** Random - the class supports random number generator-- is not required by SSF, but is implemented by DaSSF.

Table 3.2 DaSSF OTcl classes and constructor method

Class	Constructor in OTcl
Entity	Entity <name of the instance>
Event	Event <name of the instance>
InChannel	InChannel <name of the InChannel instance> <name of owner Entity>
OutChannel	OutChannel <name of the OutChannel instance> <name of owner Entity> [<write delay>]
Process	Process <name of the Process instance> <name of owner Entity> <name of the start method> <simple process or not>

3.0.2 The Implementation

3.0.2.0 The basics

The implementation introduced `Tcl_SSF_Entity`, `Tcl_SSF_Event`, `Tcl_SSF_InChannel`, `Tcl_SSF_OutChannel`, `Tcl_SSF_Process` and `Tcl_SSF_Random` C++ classes. All of them are derived classes from `TclObject` and the DaSSF C++ classes `SSF_Entity`, `SSF_Event`, `SSF_InChannel`, `SSF_OutChannel`, `SSF_Process`, and `SSF_Random` respectively. Similar to the examples in Figure 2.3 and Figure 2.4, each of the `Tcl_SSF` classes has a class derived from `TclClass` which registers the class name with OTcl. For example, `Tcl_SSF_Entity` has the following class to register "Entity" in OTcl:

```
static class Tcl_SSF_Entity_Class : public TclClass {
public:

Tcl_SSF_Entity_Class() : TclClass("Entity") {}
TclObject* create(int argc, const char*const* argv) {
    if(argc == 4)
    {
        char *name = new char [strlen(argv[1]) + 1];
        bzero(name, strlen(argv[1]) + 1);
        strcpy(name, argv[1]);
        return (new Tcl_SSF_Entity(name));
    }
    else
        return NULL;
}
} class_tcl_ssf_entity;
```

Figure 3.1 The auxiliary class of `Tcl_SSF_Entity` to register "Entity" in OTcl.

Each `Tcl_SSF` class implements its own function "int command(int argc, const*char*const argv)" which is virtual in `TclObject`. In this function, each `Tcl_SSF` class invokes the corresponding DaSSF methods at the request of OTcl through string comparison as in Figure 3.2. The methods that "int command(int argc, const*char*const argv)" function of each `Tcl_SSF` class can recognize are in Table 3.1. Since `Tclcl` will only call this command function in `Tcl_SSF` class if such method is not found in OTcl side, it is recommended those instance methods in Table 3.1 not to be overridden.

```
int Tcl_SSF_Entity::command(int argc, const char*const* argv)
{
Tcl *tcl;
tcl = &(Tcl::instance());
if(argc == 2)
{
    if(strcmp(argv[1], "now") == 0)
    {
        tcl->resultf("%f", SSF_Entity::now());
        return TCL_OK;
    }
}
```

```
    }
```

Figure 3.2 Based on string comparison, Tcl_SSF_Entity class invokes the methods of DaSSF at the request of OTcl side.

3.0.2.1 The communication between C++ objects and Otcl objects

In the invocation of DaSSF methods in the command function of Tcl_SSF classes, it is often necessary to find the corresponding object in OTcl or in C++. For example, the second argument of waitOnFor function in OTcl side is a list of InChannel OTcl objects, whose Tcl_SSF C++ object need to be located in order to make the DaSSF call. The return value of DaSSF coalignedEntities is an array of pointers to SSF_Entity objects, and they need to be converted to OTcl Objects to return from C++ side to OTcl side. The key to solve this communication is the one-to-one correspondence between a Tcl_SSF object and an OTcl object. It will never happen that there exists a Tcl_SSF object in C++ but Otcl has no knowledge of. It will also never happen that there is an OTcl object of the six classes in Table 3.1, but there is no corresponding C++ Tcl_SSF object. Each Tcl_SSF class keeps a static(i.e. there is only copy of the hashtable for all the instances of the class) hashtable to store <instance name, instance pointer> pair. The instance name is the OTcl object name. When constructing a Tcl_SSF object, the name of the object is passed to the constructor and the first thing the constructor does it to enter the name and the value of C++ keyword "this" into the hashtable. Because OTcl always communicate with C++ through strings, the hashtable enable C++ to quickly locate the Tcl_SSF object through its name. Therefore, when OTcl passed a list of InChannel objects as the second argument to waitOnFor function, C++ will be able to locate the Tcl_SSF_InChannel object pointers for all those OTcl objects in the list. Then, since Tcl_SSF_InChannel is derived from SSF_InChannel, C++ can call the DaSSF waitOnFor function after casting Tcl_SSF_InChannel object pointers into SSF_InChannel object pointers.

In addition, there is also a one-to-one correspondence between Tcl_SSF and SSF object under the fact that Tcl_SSF classes are derived from SSF classes. That is to say, the DaSSF run time systems will never construct more objects of the six classes than the SSF objects constructed with the construction of Tcl_SSF object. Therefore, when DaSSF returns an array of SSF_Entity pointers as the return value of coalignedEntities function call, those SSF_Entity pointers can be cast into Tcl_SSF_Entity pointer because we know the SSF_Entity object will only exist as part of a Tcl_SSF_Entity object. After this pointer casting, Tcl_SSF_Entity object pointer can get its name in OTcl side through the member data stored in TclObject. The name strings can be passed to OTcl side as a result of function call. In this case, if the user call "e coalignedEntities" in OTcl side, the user will get all the OTcl objects coaligned with "e" if "e" is an Entity object in OTcl side.

3.0.2.2 The naming space problem

There does exist one loophole: the hashtable does not handle the namespace at all! Should all objects be global? The avoidance of global is achieved by the "new" method provided by Tclcl package. The usage of "new" command is demonstrated in Figure 3.4 and Figure 3.5

```
Entity e
proc aproc {} {
Entity e
```

```
# troublesome, since it has the same name with the previous defined entity e
```

```
...
```

```
}
```

Figure 3.4 A troublesome program without using "new" command

Entity e

```
proc aproc {} {  
  set e [new Entity]  
  # correct since e is only an alias of an Entity object, whose name is likely to be "_o<number>"  
  ...  
}
```

Figure 3.5 a program which eliminates the problem of global objects through using "new" command.

When the new command is invoked, no object name need to be passed. The Tclcl package will create a name, which is the concatenation of string "_o" with a number, on the fly. When doing "set e [new Entity]", e is not the true name of the Entity object but only the alias of it. Each invocation of "new" command will create a unique name for any object if the user does not use the concatenation of string "_o" and a number to form a true name for any OTcl object himself or herself.

3.0.2.3 The construction of OTcl process object and the implementation limitation

One final thing is the construction of OTcl Process object. In DaSSF, a SSF_Process object with a function name passed to the constructor will execute the function whenever the SSF_Process object is active (running state as opposed to wait state) on the condition that the function contains at least one wait statement. The user has to write the function that an OTcl Process object is supposed to loop on and give the function name when requesting a OTcl Process object to be constructed. In C++, upon storing the string of the function name in its member data variable, Tcl_SSF_Process always passes the function (void (SSF_Entity::*)(SSF_Process*)&Tcl_SSF_Entity::action into SSF_Process constructor. The (void (SSF_Entity::*)(SSF_Process*)&Tcl_SSF_Entity::action function first find the OTcl function string stored in Tcl_SSF_Process object through casting the function argument - SSF_Process object pointer - to a Tcl_SSF_Process pointer. Then it does a call to Tcl_Eval with the string name of the OTcl function as the argument. Supported by Tcl core, Tcl_Eval enables the evaluation of a OTcl function or script in C.

Because of DaSSF has its own "new" and "delete" method different from c++, and TclObject and TclClass in Tclcl package are using the standard "new" and "delete" of c++, the complexity and potential memory bugs after normal DaSSF instrumentation make us decide that TclObject and TclClass not to be instrumented. Because of multiple inheritance, Tcl_SSF can not be instrumented because DaSSF requires that any base class of instrumented class also be instrumented. This limitation requires all OTcl Process objects be objects of simple processes. This in turn requires the last argument passed to OTcl Process constructor be "1" (simple is true) and the function that the OTcl Process object loops on only execute one wait statement at any one loop and the wait statement to be the last statement it executes in the loop.

4.0 Conclusion and Future Work

With the goal of supplying DaSSF with the OTcl APIs, we actually touched the area of object reuses, objects in one language (c++) wished to be reused in another language (Tcl) with minimum effort. However, our attempt has never been to write a pseudo-c++ tcl interpreter. We are unsure if it can be done. As a result, the objects created by the OTcl DaSSF APIs are with their limitations stemming from OTcl extension and Tclcl package, to name a few: every method and every member data is public; instances of the same class could have different member methods and member data. Nonetheless, the idea of objects in OTcl is able to simplify software development in Tcl.

With the limitations of only supporting simple processes and having to use "new" to eliminate the problem of global objects, the goal of supplying DaSSF with OTcl APIs are achieved. Future work may concentrate on supporting non-simple processes. In our view, it could be both more work to accomplish the task of object name space and less useful than supporting non-simple processes since the use of "new" command has already largely eliminated the problem of global objects. The first step to support non-simple processes is to be able to rewrite TclClass and TclObject classes in Tclcl package in order to instrument them.

5.0 Acknowledgements

This thesis is based on the development of Tclcl package of ns community in University of California at Berkeley. We thanks the help from ns community, especially in tracking down the bugs with OTcl. We also give thanks to Jason Liu and Brian J. Premore at Dartmouth College for their suggestions and kindly support throughout this thesis.

Reference:

- [1] Nicol, D.M. and Liu, J.(1999) **DaSSF (Version 1.0.3)** [Computer Software]. Dartmouth College, NH
- [2] McCanne, S., et al. **NS (Version 2.1b5)** [Computer Software]. University of California, Berkeley
- [3] Nicol, D.M. and Liu, J. **DaSSF: Dartmouth Implementation of SSF** (Dartmouth College, Department of Computer Sciences, March 1999)
- [4] Brent B. Brench, **Practical Programming in Tcl and Tk**, Prentice Hall, New Jersey. 1995.
- [5] Wetherall, D., and Lindblad, C.J. **Extending Tcl for dynamic object-oriented programming**. In *Proceedings of the Tcl/Tk Workshop* (Ontario, Canada, July 1995).
- [6] McCanne et al. **Toward a Common Infrastructure for Multimedia-Networking Middleware**, *Proc. of the 7th Intl. Workshop on Network and Operating Sys. Support for Digital Audio and Video (NOSSDAV '97)* (May 1997).

Appendix Examples of using DaSSF OTcl APIs:

HelloWorld.tcl

```
# helloworld.tcl
# this program created one entity with two processes communicating with each other , one send and the other receive.
Class helloWorld -superclass Entity
set DELAY 20
set rcvd 0
Entity hi
InChannel IN hi
OutChannel OUT hi $DELAY
OUT mapto IN
hi proc send { } {
```

```
global OUT
global DELAY
OUT write [new Event]
hi waitFor $DELAY
}
```

```
hi proc rcv { } {
global IN
global rcvd
set rcvd [expr $rcvd+[length [IN activeEvents]]]
hi waitOn IN
}
```