CSLUsh: AN EXTENDIBLE RESEARCH ENVIRONMENT

Johan Schalkwyk, Jacques de Villiers, Sarel van Vuuren and Pieter Vermeulen

Center for Spoken Language Understanding, Oregon Graduate Institute of Science and Technology, 20000 N.W. Walker Road, P.O. Box 91000, Portland, OR 97291-1000, USA

ABSTRACT

The CSLU shell (CSLUsh), is a collection of modular building blocks which aim to provide the user with a powerful, extendible, research, development and implementation environment. Implemented in C with standardized Tcl/Tk interfaces to provide a scripting and visualization environment, it allows a flexible cast for both research algorithms and system deployment. This shell is the architecture on which the CSLU Toolkit is built and may be downloaded for non-commercial use from http://www.cse.ogi.edu/CSLU/toolkit.

1. INTRODUCTION

With the advent of the information age, it has become increasingly important for researchers and developers of human language technologies and other communities to be able to share ideas and technology. Sharing of actual algorithmic implementations has been limited due to the wide variety of computers and operating environments used around the world. In a multi-disciplinary field such as human language technology, it is advantageous to researchers with differing backgrounds to share results in a timely fashion and comprehend the interactions of recent advances.

A single transparent research and implementation tool would stimulate research by creating an environment in which ideas could be shared by the exchange of code without consideration of issues such as cross platform portability.

Our solution to this problem built on previous efforts [1], was to implement real-time systems and to let researchers share in, and incorporate the latest advances effortlessly. Great care was taken to design all of the core components to operate in as efficient and consistent a manner as possible, with special attention given to modularity, portability and extendibility. In addition, implementation considerations such as pipelining, networked sharing of computing and input/output resources for real time systems have been addressed.

Implemented in Tcl/Tk [2] and C, CSLUsh supports a wide range of research activities, including data capture and analysis, corpus development, research in multi-lingual recognition and understanding, dialogue design [3], speaker recognition and language identification, among others. CSLUsh has also been used to implement real-time speech recognition systems capable of handling multiple telephone conversations [4].

In this paper we describe the architectural foundation of the CSLU shell. Section 2 describes the software architecture. Section 3 discusses each of the core components.

2. SOFTWARE ARCHITECTURE

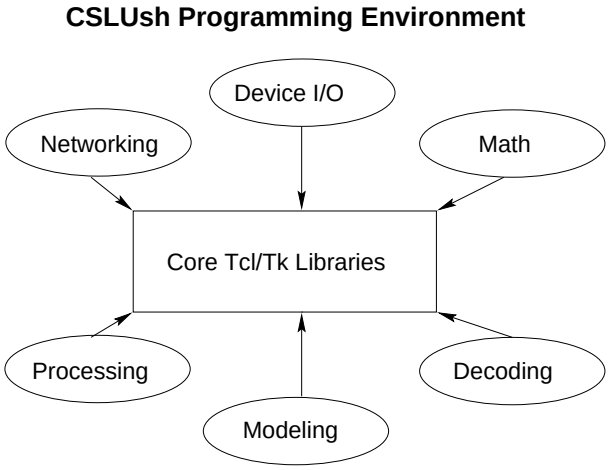


Figure 1. Software Architecture of the CSLUsh.

Figure 1 presents the software architecture of CSLUsh. Its foundation is a set of efficient, and where necessary, pipelined C libraries (CSLU-C) for the functions which support the basic algorithmic operations and associated utilities. These libraries can be used directly with a documented C API to build applications.

The extendible scripting language Tcl is used to access functionality in an efficient scripting environment, by creating core components (Tcl packages) which "glue" the basic operations according to a well defined API. Individually, each core component extends the usability of CSLUsh by providing specific capabilities such as, for example, networking or matrix manipulation. Collectively these building blocks form an environment in which one can with ease plug and play various components in order to design, debug and execute complex algorithms. As with Tcl, CSLUsh also provides error messages, command stack tracing and exception handling.

The packages are dynamically loaded as needed, providing a small footprint for implementation. This mechanism also allows for easy alteration and extension by a user. Following

the documented API, a different version of an existing algorithm or a new functionality can be separately compiled and will seamlessly become part of CSLUsh. The standard Tcl version control mechanism allows for managing these changes and additions.

Although the software has a strong emphasis on spoken language technology many of the core components are useful in other domains as well.

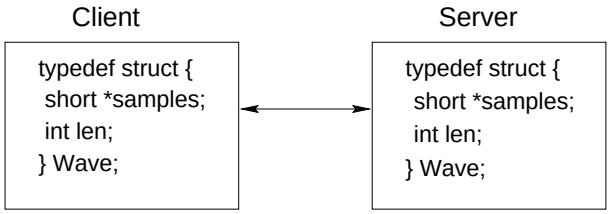
3. CORE COMPONENTS

Networking

The Toolkit was developed in an environment which makes heavy use of distributed computing. CSLUsh extends Tcl to provide client/server communications (e.g. telephony and TTS) and facilitate optimal data and code passing for remote execution of scripts. Servers are managed by a central registry daemon.

Input and output data are treated as generic objects, which can be automatically and efficiently transported across the network and saved to and loaded from disk. The objects are conventional C data structures represented at the CSLUsh level as simple string identifiers. Byte format conversions and memory management are handled transparently and networking is optimized to fit the object size.

Figure 2 presents an example of this process. The client reads a speech wave file from disk and sends it to the server. Each time the server receives a wave object it calls the `handleWave` procedure, which in this example prints out some information regarding the received wave object, using the `wave info` command.



Client

```
context link client mrburns 9999 secret
set w [wave read test.wav]
client send $w
```

Server

```
proc handleWave {server clientID cntr obj} {
  puts [wave info $obj]
}
context tcp server -port 9999 -password secret
server container wave handleWave
```

Figure 2. Client/server communication.

This generic approach to data handling (object serialization) allows complex algorithms to be trivially run in a distributed manner with different steps of the computation occurring on different machines. This example will run across machines with different operating systems and byte order.

Device I/O

Device input/output builds upon the generic data objects utilizing an object oriented design, which allows for the design of consistent interfaces to a wide variety of input and output devices such as the PC speaker, a voice modem, and a text-to-speech (TTS) engine.

The following example creates a new TTS object and attaches to any TTS server on the network (i.e. one that has been registered with the central registry daemon). Next an audio object is created and attached to the audio device on the local machine. This could also have been a telephone line interface. The application is shielded from the differences between such audio devices.

```
TTS create text2speech "TTS Demo"
Audio create desktop "Audio Demo" {hostname local \
    type audio}
```

Finally the `Wave` event output of the TTS object is connected to the `Wave` event handler of the audio device. Once these connections have been made any audio output generated by the TTS object will be sent to the speakers.

```
text2speech addWave-> desktop <-Wave
text2speech <-Text "Hello, world."
```

Note that this mechanism provides for non-blocking asynchronous communication with the various devices, allowing real-time implementation of input-driven systems.

Math

Matrices (**Array** objects) are used extensively throughout CSLUsh for input and output results. For example feature processing and modeling make extensive use of often huge matrices. To allow assimilation, partitioning, transformation and inspection of these matrices the CSLUsh development environment includes a math module. Named Mx, this module provides extensive scripting capability for matrix and vector math.

In addition to the usual math operations (real, complex, element-wise, ranging etc) Mx allows operations on lists such as

```
set x [mx join row [list $a $b $c]]
```

One of the powerful features of Mx is that it allows the programmer to explicitly control memory usage. Advantages of this include considerable execution time speed-ups and a generally small memory footprint.

As an example of such functionality, consider the equation

$$r = \mathrm{cov}(y)\mathrm{cov}(x)^{-1},$$

where x and y are data matrices, and "cov" is the covariance. Furthermore, suppose that this equation will be called many times in a loop so that considerable speed-up may be gained by exploiting the memory usage of temporary variables. The following code fragment details how this may be done using Mx.

```
mx cov $x S_x
mx cov $y S_y
mx cholinv $S_x Si_x
set r [mx prod $S_y $Si_x]
```

Here the suffix position for the temporary variables S_x, S_y, and Si_x indicates to Mx to allocate memory for them only at the first pass of the loop and on subsequent passes to simply overwrite their contents. A prefix position on the other hand indicates to Mx not to reuse memory – as for the variable r in the above example. This implementation allows execution speed to approach that of dedicated C-code. To change the above code fragment to ignore the first 2 rows of S_y in the product it suffices to specify a subrange

```
set r [mx prod $S_y.(:2,:) $Si_x]
```

Other CSLUsh modules can inherit Mx 's characteristics through the CSLU-C API.

Speech Processing

Signal processing includes support for the usual feature processing routines such as PLP, MFCC and LPC. It also includes their delta and Rasta derivatives. All processing routines are fully pipelined and therefore easily cascaded. The pipelining allows for real time processing of speech in systems; i.e. using the time the current input is being created to process previous chunks of it, instead of waiting for the phrase to complete. The example below illustrates this concept by filtering each of the linear predictive cepstral coefficients (LPC) using a Rasta fitler.

```
set w [wave read test.wav]
set lpc [analysis lpc initialize]
set wlpc [analysis lpc $lpc $w]
set rasta [analysis rastafilter initialize 12]
set wrasta [analysis rastafilter $rasta $wlpc]
```

Work is in progress on integrating a new feature processing module into the Toolkit. This module extends existing features to include pitch extraction, cepstra, filter bank outputs and frequency selective versions of PLP and MFCC. Another module provides robust speech/non-speech detection by adaptively tracking the noise floor in a speech signal. A further module provides uni- and multi-dimensional FIR filtering (in time and frequency) as well as LDA-based analysis [5].

Modeling

Modeling includes neural networks (classifiers and regression models), vector quantization, gaussian mixture modeling and hidden Markov modeling (CSLUhmm).

These modeling techniques are to a large extent interchangeable. For example the embedded reestimation algorithm provided by the main HMM library may also be used to reestimate neural network targets [6]. Parameter tying may be done at either the model, state, mixture component, mean and/or covariance level. The HMM package also supports decision tree state clustering and triphone synthesis to provide a framework in which to build full context dependent recognizers.

```
# Vector quantization using the LBG algorithm
# with iterative cluster splitting

set gvqob [gvq configure -iter $iter -mix $mix]
gvq initialize $gvqob vq
foreach speaker $speakerlist {
  while 1 {
    foreach wav $wavfiles($speaker) {
      set data [feature $wav]
      gvq lbg::accumulate $gvqob $data vq
      nuke $data
    }
    if {![gvq lbg::update $gvqob $vq]} break
  }
  obfile write $fob $speaker $vq
  gvq reset $vq
}
```

Figure 3. Training multiple vector codebooks.

Throughout the design and implementation of these modeling techniques we took care to minimize resource usage. To build large systems with limited resources we took care to optimize both memory and execution time requirements. Based on these considerations a typical training session may be conceptually scripted as:

```
model configure
loop models:
  model initialize
  loop files:
    model accumulate
  end
  model update
  model save
  model reset
end
```

The typical procedure of initialize, accumulate, update, save and reset allows all model parameters and training variables to share the same memory during training. The static memory footprint is thus dependent only on the size of the largest model to be trained with the dynamic memory determined only by feature computation. The code fragment in figure 3 illustrates this process by computing several vector codebooks and saving them to disk in a machine-independent format using the generic object interface described earlier.

HMM models are created and configured using CSLUhmm configuration scripts. Figure 4 presents an example. In this example monophone models are created for the phonemes /w/, /ah/, /n/, /sil/ and /sp/. The short pause model (/sp/) is then tied to the center state of the silence model (/sil/). The HMM configuration language provides a mechanism in which one can specify new HMM models and also edit existing HMM models. The collection of such scripts

documents the process of building a recognizer in an easily readable and understandable format.

```
prototype mono numstate 5 mixtures 3 transp
  0.000  1.000  0.000  0.000  0.000
  0.000  0.600  0.400  0.000  0.000
  0.000  0.000  0.500  0.500  0.000
  0.000  0.000  0.000  0.600  0.400
  0.000  0.000  0.000  0.000  0.000;

prototype onestate numstate 3 mixtures 3 transp
  0.000  0.500  0.500
  0.000  0.500  0.500
  0.000  0.000  0.000;

define mono <w> <ah> <n> <sil>;
define onestate <sp>;

tie <sil>.state[2]  <sp>.state[1];
```

Figure 4. Hmm Configuration.

All of the above mentioned functionality is integrated within the CSLUsh environment. A stand-alone application such as HMM embedded training is therefore just another CSLUsh script which reads a set of predefined input files and performs embedded training. The user can with relative ease change such applications to meet specific needs, rather than comply with the predefined interface. For example rather than computing features and then having the training scripts read these feature files, the standard training scripts can be altered to compute the features on the fly.

Further technology being integrated within the CSLUsh framework includes vocal-tract normalization (VTN), maximum a-posterior (MAP) training, speaker adaptive training (SAT), and maximum likelihood linear regression (MLLR) adaptation. Currently these algorithms are being used to build the OGI large vocabulary recognizer and the OGI speaker recognition system.

Decoding

Decoding incorporates Viterbi decoders for word spotting and finite state grammars, within the CSLUsh framework.

Currently under development is a three pass (Forward, Backward, A*) search [7] which works with HMMs and Neural Network hybrids interchangeably. In this implementation the Backward/A* search is used as a framework in which to incorporate N-gram language models. For large vocabulary tasks (5k $-$ 65k words) we are also working on a dedicated decoder based on pronunciation prefix trees.

4. CONCLUSIONS

Several applications have been developed with the CSLUsh programming environment. The CSLU rapid prototyper (CSLUrp) is a graphically-based authoring tool built on top of CSLUsh that enables the iterative design and immediate testing of spoken dialogue systems. The Toolkit also includes a display tool Lyre, which provides basic browsing capabilities by extending the underlying Tk widgets with a waveform display, a generalized spectrogram display and a label display widget.

Our hope in releasing this Toolkit is to engage a large number of people in a participatory design to create and donate increasingly more powerful tools.

The Toolkit is released with source code so that any who desire can improve it. We hope that new users will bring new perspectives and new capabilities to the Toolkit, and that these improvements will be shared with us and others so that all may benefit.

ACKNOWLEDGMENTS

A large group of people have contributed and are continuing to enhance this Toolkit guided by the vision of Ron Cole who initiated this effort and Mark Fanty who currently manages the Toolkit project. Much of the initial speech functionality was inherited from the OGI speech tools authored by many including Mark Fanty and Fil Alleva. The inputs, algorithm contributions and suggestions of the users at OGI have been extremely helpful.

This work is co-sponsored in part by ONR and ARPA grant xxxx and CSLU center members.

REFERENCES

[1] M.Fanty, J.Pochmara, and R.A.Cole, "An interactive environment for speech recognition research," *Proceedings of the International Conference on Spoken Language Processing*, October 1992.

[2] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[3] S.Sutton, D.Novick, R.Cole, P.Vermeulen, J.deVilliers, J.Schalkwyk, and M.Fanty, "Building 10000 spoken dialogue systems," *Proceedings of the International Conference on Spoken Language Processing*, October 1996.

[4] R.A.Cole, D.G.Novick, P.J.E.Vermeulen, S.Sutton, M.Fanty, L.F.A.Wessels, J. Villiers, J.Schalkwyk, B.Hansen, and D.Burnett, "Experiments with a spoken dialogue system for taking the u.s. census," *Free Speech Journal, http://www.cse.ogi.edu/CSLU/fsj/html*, vol. I, 1997.

[5] S. van Vuuren and H.Hermansky, "Data driven design of rasta-like filters," *this proceedings*, 1997.

[6] Y.Yan, M.Fanty, and R.Cole, "Speech recognition using neural networks with forward-backward probability generated targets," *Proceedings of the International Conference on Acoustic, Speech and Signal Processing*, vol. IV, pp. 3241–3244, 1997.

[7] F.Alleva, X.Huang, and M.Hwang, "An improved search algorithm using incremental knowledge for continuous speech recognition," *Proceedings of the International Conference on Acoustic, Speech and Signal Processing*, vol. II, pp. 307–310, 1993.