

Examensarbete 15 poäng C-nivå

A SPEECH RECOGNITION SYSTEM FOR PEIS ECOLOGIES

Reg.kod: Oru-Te-DT3017-D107/08

Jonas Ullberg

Dataingenjörsprogrammet 180 p

Örebro höstterminen 2008

Examinator: Alessandro Saffiotti

Örebro universitet
Institutionen för teknik
701 82 Örebro



Örebro University
Department of Technology
SE-701 82 Örebro, Sweden

Abstract

This report summarizes the assignment and solution to the task of developing a speech recognition infrastructure for an intelligent home environment running the PEIS ecology middleware. Such a system is useful since it allows humans to interact with the PEIS ecology environment in a natural way.

The speech recognition system was implemented as three standalone modules, which handle audio input, speech recognition and device actuation respectively. One of these modules also presents a new approach to performing lightweight tasks within the PEIS ecology through the use of the Lua scripting language. The task was assigned as a 15 point exjob by the AASS research lab at Örebro University to one student as a final project for the bachelor level degree.

Contents

Abstract	1
Contents	2
1. Introduction	3
Background	3
PEIS Ecology	3
Problem & Requirements	4
2. Choice of libraries.....	5
Speech recognition engine	5
CMU Sphinx.....	5
HTK/Julius	5
Dispatcher / Scripting language	5
3. Design and implementation.....	7
PEIS ADIn Tool.....	7
Usage	7
PEIS Julius	8
Usage	8
Grammar.....	9
PEIS Lua.....	10
Usage	10
Implementation details	11
Example	11
PEIS specific variables.....	13
PEIS C bindings	13
PEIS auxiliary library	15
4. Conclusion.....	16
Limitations and problems.....	16
Future work	17
5. References	17

1. Introduction

Background

PEIS Ecology

The PEIS ecology project [1] is a research project at AASS mobile robotics lab at Örebro University devoted to exploring how different physically embedded intelligent systems (PEIS) can interact with each other to produce an intelligent home environment for the future. The heart of the PEIS ecology project is the PEIS kernel [2] which is a middleware that each device in the PEIS environment uses to share information with other devices in the ecology, this is done by reading and writing key/value pairs in a distributed tuplespace. A computer program that uses the PEIS kernel is called a component and one of the project's main ideas is that the intelligence of the environment should be realized by the collaboration of many small components providing simple services. This contrasts with the more established tendency in robotics of constructing fewer more complex and multi-purpose autonomous robotic devices.

As an example one might consider a simple vacuum cleaning robot (e.g. "Roomba") that is available for household use today. To be affordable these kinds of robots are typically relatively simple in their construction, using only bumpers for collision detection etc., which makes them less autonomous than their more expensive counterparts. However, integrating these kinds of robots into the PEIS ecology would provide them with access to additional information that could help them perform their task, as for an example by querying a webcam mounted in the ceiling for detected obstacles or by communicating with each other to coordinate the task as a whole.

The concept of PEIS is not limited to only including what we typically refer to as robots; it could also include a fridge or something as simple as a lamp if they have some means of communication.

Problem & Requirements

Since the PEIS ecology project has previously lacked a speech recognition system – which could be considered to be an integral part of an intelligent home environment – the goal of the project is to construct a system that can actuate the PEIS environment in response to spoken commands from a user. Such a system is useful since it allows humans to interact with the PEIS ecology environment in a natural way. It might also provide physically impaired people with another degree of freedom, allowing them to perform common tasks such as turning on the lighting in the room.

The speech recognition system has two main tasks that it should perform. First, it performs a speech-to-text translation of spoken sentences. Second, it actuates the PEIS environment in response to recognized commands. The system will make use of an existing speech recognition engine (SRE) to handle the first task while the actuation will be handled by a custom solution. The SRE to use is chosen in the beginning of the project, the choice is however limited to open source alternatives. The fact that the SRE have not been chosen in advance puts some limitations about what can be required of the final system in terms of functionality though, since it will be limited to the capabilities of the SRE and how well it can be integrated into the PEIS environment. To put it differently, this project is of an "open ended" nature since it is not known in detail what can be achieved in practice. There are however some key requirements that should be addressed by the system.

From a development perspective the speech recognition system is required to be user friendly and modular. In this project the term user friendliness covers the central topic of how the speech recognition system should be configured to actuate other components and relates more to the person that configures the system than the end-user. The question is important since the PEIS ecology is an ongoing research project and new components are constantly being created. Thus it was decided that the speech recognition system should be able to control any foreseeable component without requiring any modifications to its source code. This will enable the system to carry out both low and high level tasks in the ecology. An example of a low level task is telling an autonomous robot to turn left, while a high level task is telling the same robot to retrieve a drink from the fridge and deliver it to the living room.

Another requirement that relates to user friendliness is that the system should be able to recognize sentences spoken by different users in the environment without requiring prior training, in other words the system should be speaker independent. To be fully PEIS-compliant the speech recognition system should also be modular in the sense that each reusable part of it is realized as a separate PEIS component. Modularity and reusability are two important concepts in the PEIS ecology vision since it allows components to work together, this in turn will make it possible for the ecology to attain an ambient intelligence, which is one of the projects main goals.

During the course of the project one additional task will be to determine what can and cannot be done with respect to the constraints in the PEIS middleware, as an example the PEIS middleware does not support streams so there can be no real time audio streaming. Another problem is deciding how different actions in the ecology should be associated with the recognized commands. Since a lot of problems still remain to be resolved in the field of speech recognition the project restricts itself to the recognition and execution of simple voice commands from a predetermined set of sentences (i.e. command-and-control).

2. Choice of libraries

Speech recognition engine

At the time of writing there seemed to be two major open source speech recognition engines (SREs) that stood out from the rest, namely CMU Sphinx and HTK/Julius, where CMU Sphinx appeared to be the most popular one. Both of these libraries are supported by Voxforge [3] which is a community driven effort to bring free speech recognition to the open source world. Voxforge does this by collecting transcribed speech audio which is used to train acoustic models that are published on their homepage under the GPL license.

It was necessary to use the acoustic models from Voxforge for this project because creating one from scratch would have required many hours of transcribed speech.

CMU Sphinx

CMU Sphinx [4] is a SRE developed by the Sphinx group at Carnegie Mellon University with a focus on command-and-control type applications. Sphinx seems to be the main choice of SRE in the English speaking open source world today; being used for voice control in GNOME and in the Player project amongst others. With these facts in mind one might think that Sphinx would be the obvious choice for this kind of project. Unfortunately early testing gave the impression that it would take too long to learn how to use the SRE in practice.

HTK/Julius

Julius [5] is a Japanese research project developed by the Interactive Speech Technology Consortium (ISTC). Unlike Sphinx; Julius focuses on handling dictation tasks and not command-and-control. Another difference is that its training is done with the third party HTK toolkit whereas Sphinx includes a trainer. Julius was chosen for this project since it worked more or less “out of the box” when using the models from Voxforge. Julius also had several choices for audio input and was more lightweight than CMU Sphinx when comparing the size of the engine and models. Julius’ biggest drawback was its lack of good English documentation. This problem was however alleviated by some good tutorials found on Voxforge.

Dispatcher / Scripting language

To actually make something happen when a sentence got interpreted it was necessary to implement some kind of component that could listen for recognized speech tuples and then actuate other components in response to the recognized command. We call this module a “dispatcher”.

Writing a dispatcher turned out to be a tradeoff between ease of use and flexibility when a couple of solutions were considered. The most basic dispatcher would at least have to be able to listen for new recognized speech in the PEIS network and then compare its string representation to some predetermined set of known commands, e.g. “OPEN FRIDGE”. If a command was recognized the dispatcher would then have to retrieve the id of the components to be actuated to be able to set its tuple values.

One simple dispatcher solution that was considered involved an XML file containing sentences to listen to and their source path, the way to identify the id of the target device and finally the tuple name and a new value of the target tuple on this device. Such an XML document could have looked something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rules>
  <rule name="open fridge door">
    <antecedent id="*" key="*.command.*" value="OPEN FRIDGE" />
    <consequent locateid="*" locatekey="kernel.name"
      locateval="fridgedoor" key="requested-state" value="opened" />
  </rule>
  <!-- ...More rules... -->
</rules>
```

The above XML solution seemed overly verbose considering its lack of extensibility. For example, it would be hard to extend it to take advantage of any logical structure in the incoming commands or to forward parts of the recognized speech to the target tuple. A custom plain text approach could have been more readable but it would still have shared most of the problems inherent in the XML solution.

Seeing the drawbacks with a static rule based approach one feasible alternative seemed to be to create a custom script-like language capable of handling a bit more advanced tasks than the XML solution. The idea was that the imagined language would have a very simple syntax for performing queries in tuplespace which could have been implemented with the help of lexical analyzer and a parser generator such as LEX and YACC. When considering this idea further it was however concluded that it wouldn't be possible to implement this within the limited timeframe of the project.

Since both of the previously two mentioned approaches had serious drawbacks it was concluded that to be able to meet the needs of expressiveness without making the size of the project get out of hand something else had to be tried.

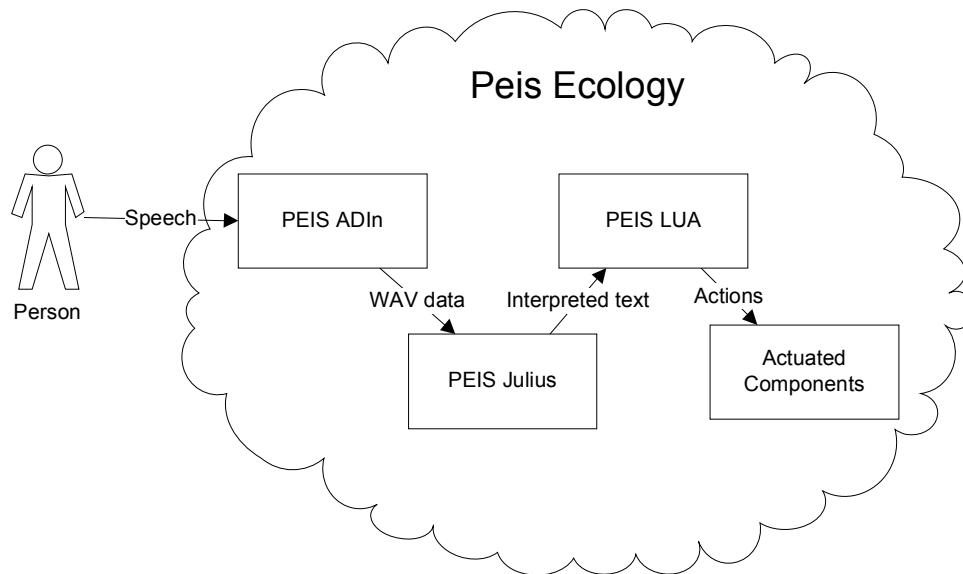
As it turned out a good solution seemed to be to make PEIS bindings for the scripting language Lua [6] which is described as a "powerful, fast, light-weight, embeddable scripting language", "ideal for configuration, scripting and rapid prototyping". By using an embedded scripting language for the dispatching task a whole lot of benefits were gained, most notably the ability to add context to the interpretation through the use of variables and conditionals (e.g. the system knows that we are talking about a robot and can use that information to resolve the meaning of a sentence such as "turn left").

Lua was chosen for this project because of its small size and the way in which it easily integrates with C. An alternative among embeddable scripting languages in widespread use today is Python. Python which is inarguably a more common choice than Lua was however deemed to be more of a general-purpose programming language with script-like capabilities, which wasn't necessary or even desired when solving such a simple task as dispatching.

There were probably other scripting languages that could have accomplished this task as well and in addition to Python, no in-depth comparison was performed to determine that Lua in fact was the best choice. Nevertheless Lua had some key benefits like its size being less than 100 KiB excluding the standard libraries and having good performance along with a small memory footprint.

3. Design and implementation

This part of the report describes the components that were written for this project. Three components were written; the first one is named “PEIS ADIn” and is used to record audio and make it available within tuplespace. The second component is called “PEIS Julius” and is responsible for interpreting what has been said in the audio and to make the result of the analysis available to the rest of the ecology. The final component is called “PEIS Lua” and provides a simple scripting interface to PEIS that can be used to interact with other components in response to the result from the speech interpreter.



PEIS ADIn Tool

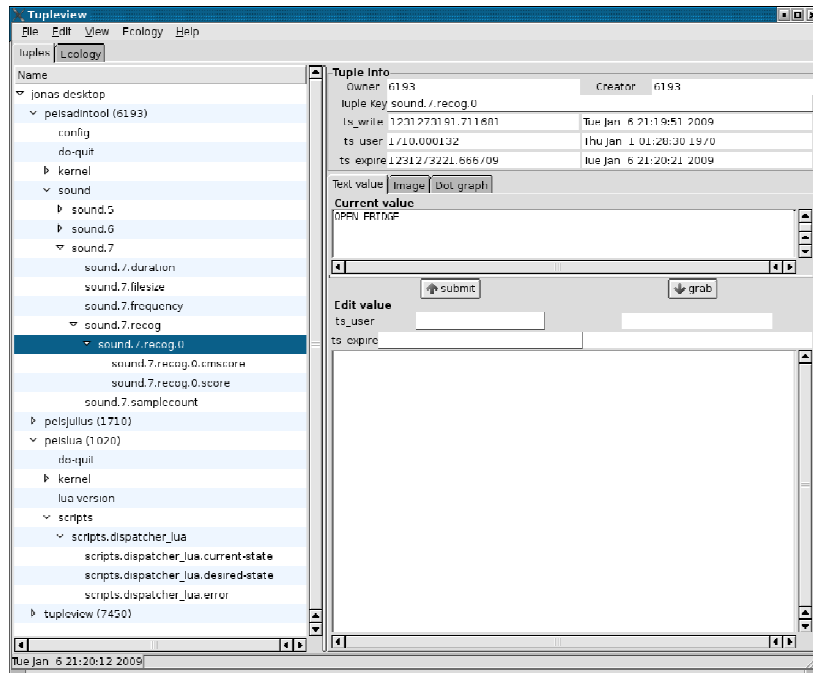
The PEIS ADIn Tool is a tool able to record audio from a microphone and publish it as uncompressed WAV files into the PEIS ecology. The tool is an extension of the ADIn tool that ships with Julius and retains all original capabilities of it while it adds the PEIS output option. In this project it is used to perform audio segmentation on the microphone input stream by listening for continuous high sound activity while segmenting out the silence.

Usage

If the tool is started with the command line “`peisadintool -in mic -out peis`” it will initialize the microphone and start publishing the recorded audio using default parameters for sampling rate etc. As previously mentioned the tool keeps its original functionality as well so if the flag “`--help`” is given as a command line option the user will be presented with a list of options along with their corresponding flags. If the ADIn tool is unresponsive it might be because the input level is too low, this can be remedied by using the Linux tool “`alsamixer`” to increase the microphone boost.

PEIS Julius

PEIS Julius is the actual speech recognition engine component; it works by listening for uncompressed WAV files in tuplespace which it interprets. If the WAV file contains a recognized sentence PEIS Julius decorates the sound file with the result of the interpretation. It does this by adding additional sub keys to the key containing the sound. How this works in practice is best explained with an example:



The image above shows a screenshot of the “Tupleview” tool that is frequently used in the PEIS ecology to observe and modify the values of tuples in the ecology. The listbox to the left shows some of the tuples being used by the PEIS ADIn tool. The tuples containing the audio are the ones named “sound.*”, e.g. “sound.7”. Each of these contains additional tuples that give information about file size etc. In this image “sound.7” has been decorated with the result of a successful interpretation, this can be seen by looking at the highlighted key named “sound.7.recog.0” which contains what has been said. Note that this tuple didn’t exist until PEIS Julius interpreted the sound; it was added by PEIS Julius to the ADIn tool’s namespace after the successful interpretation.

Usage

PEIS Julius doesn’t take any special command line arguments except for those that control the PEIS kernel. When the engine is started it reads a configuration file from the relative path “./lang_mod/peis_julius.jconf”. The configuration file contains the path to the grammar along with a wide range of different options that specify how Julius will perform the recognition and behave in general. Many of the settings found in this file require knowledge about speech recognition and Hidden Markov models to be changed, but the default settings work well so it isn’t necessary to change any of those. There is also an option to specify the input source but this one should always be set to “-input mic” because PEIS Julius will always read tuples from tuplespace and the mic input option has some desired side effects that is necessary for the program to work correctly. One option that can be of interest however is the “-output” option which specifies how many sentence hypotheses Julius will try to find. All found hypotheses will be published in tuplespace sorted by score (e.g. the interpretation sound.7.recog.0 is more likely to be correct than sound.7.recog.1).

Grammar

Example.grammar

```
S : NS_B CMD_LOOP NS_E
CMD_LOOP : CMD_LOOP THEN CMD
CMD_LOOP : CMD
CMD : MOVEMENT_ACTION DIR
CMD : MOVEMENT_ACTION DIR SPEED
```

Example.voca

```
% NS_B
<s>          sil

% NS_E
</s>        sil

% THEN
THEN         dh eh n

% MOVEMENT_ACTION
MOVE         m uw v
TURN         t er n

% DIR
FORWARD     f ao r w er d
AHEAD       ax hh eh d
BACKWARDS   b ae k w er d z
LEFT        l eh f t
RIGHT       r ay t

% SPEED
SLOW        s l ow
FAST        f ae s t
```

To be able to use (PEIS) Julius one must first define a grammar. The grammar is built up by a .voca and a .grammar file, an example of such a pair is shown in the table above. The .voca file contains words divided into categories along with their pronunciation and is used by the .grammar file which defines valid sentences.

Word categories are defined in the .voca file by using a % as the first sign followed by the name of the category. Once a category is defined the underlying rows should contain the words that belong to it along with their pronunciation which is constructed out of phonemes from the word list that have been used to train the acoustic model. The special word categories “NS_B” and “NS_E” define silence in the beginning and ending of a sentence and their containing “sil” phonemes are used as delimiters by Julius and shouldn’t be changed.

The .grammar file is somewhat similar to the .voc a file. Each line in the .grammar file defines a production rule which is built up by a name, a “:” delimiter and then any number of other production rules or word categories that it is satisfied by. The “S” rule is special because it is the entry point for Julius when it tries to match a sentence and it should therefore always be defined. As can be seen in the example it is also possible to define recursive or overlapping rules.

Once the two grammar files have been written Julius grammar compiler mkdfa.pl should be run to convert them into a format suitable for the engine. The example above would be compiled with the command line “mkdfa.pl Example” leaving out the file endings; this would result in compiled grammar that could interpret sentences as “Turn left” or “Turn left then move ahead slow”. A more careful examination of the grammar reveals that it also allows sentences like “Turn backwards”

which doesn't make much sense, this is however not believed to be a big issue since if such a sentence gets interpreted it's either because of the fact that the user actually said it, which would make the interpretation correct, or because the speech recognition engine made an error which would imply a problem with the recognition itself and not the grammar. In other words it's not necessary to constrain the grammar too much. If one wants to list some random sentences that are acceptable by the grammar it can be done with Julius generate tool (i.e. "generate Example").

PEIS Lua

PEIS Lua is the name of the component responsible for running tasks in response to interpreted commands; it does this by running Lua scripts with a small library extension containing PEIS bindings. The component is able to run many scripts in parallel which can be started, stopped and paused individually through tuplespace. The reason for making the dispatcher able to run many scripts at once instead of having one instance for each script is that the dispatching tasks/scripts were conceived to be more lightweight than an ordinary PEIS component. Another thing worth mentioning is that although the main intention with PEIS Lua was to solve the task of dispatching, it can be used for a lot of other common tasks as well.

Usage

PEIS Lua is started with a command line of the form "peislua [peis kernel options] [script1.lua script2.lua]". The only command line option which affects PEIS Lua itself is the optional "--run" flag which specifies that all subsequent scripts should be loaded in a running state. The default is that the scripts are loaded but not started.

The following table shows which tuples are being used to control an imagined script called test.lua. The keys are the same for all scripts except for the "test_lua" part which would be replaced with each script's corresponding filename (with the "." in the filename replaced with a "_").

<u>Tuple</u>	<u>Description</u>
scripts.test_lua	Contains the source code of the script, can be changed while the script is stopped. Changes will be written to the physical file.
scripts.test_lua.current-state	Contains the current state of the scripts, either "stopped", "running" or "paused".
scripts.test_lua.desired-state	Setting this tuple changes the state of the script, valid values are "stopped", "running" or "paused"
scripts.test_lua.error	If an error occurs in a script which causes it to terminate this tuple contains an error message. E.g. "[string "dofile("peis_util.lua")...]":6: Unexpected symbol near '3'", which is a syntax error.

Implementation details

Because PEIS Lua can run many scripts at once, and they all share the same PEIS kernel, there were some noteworthy problems.

It would have been good if each script could manage its own subscriptions but since they all share the same kernel this could possibly make the scripts interact with each other in an undesired way. The solution was to make the main PEIS Lua component subscribe to the entire ecology during startup so that each script could access all tuples in the ecology without subscribing to them. One downside with this approach is that it generates a lot of unnecessary network traffic but it has an advantage in the fact that the scripts don't have to wait for their subscriptions to propagate when they are started.

Another problem was implementing the callback functionality and preemption. Lua terminates a script by raising an error which involves a long jump within the embedding C code; this didn't work well with the PEIS bindings since they were written in C++ and the stack didn't get unwound correctly resulting in memory leaks. The solution to this was to avoid using C++ objects that allocate dynamic memory (e.g. `std::string`) in the bindings.

Example

```
dofile("peis_util.lua");

spd = {0, 0, 0};
old_spd = {0, 0, 0};

function onrecog(id, key, val)
    local function set_speed(fwd, dummy, rot)
        spd = {fwd, dummy, rot};
        local id = locate("kernel.name", "peisplayer");
        if id then
            set_tupleval(id, "position.0.setvel", make_list(unpack(spd)));
        end
    end

    local actions = {};
    actions["MOVE FORWARD"] = function () set_speed(0.2, 0, 0); end
    actions["MOVE BACKWARDS"] = function () set_speed(-0.2, 0, 0); end
    actions["TURN LEFT"] = function () set_speed(0, 0, 0.5); end
    actions["TURN RIGHT"] = function () set_speed(0, 0, -0.5); end
    actions["STOP"] = function () old_spd = spd; set_speed(0, 0, 0); end
    actions["ABORT"] = function () old_spd = spd; set_speed(0, 0, 0); end
    actions["CONTINUE"] = function () set_speed(unpack(old_spd)); end

    if(actions[val]) then
        actions[val]();
    end
end

add_callback(-1, "sound.*.recog.0", onrecog);
loop_callback();
```

The table above shows a Lua script that can be used together with the example grammar presented earlier to control a robot. It would be outside the scope of this report to give an in depth description of Lua, but the language is not very different from languages like Python and should be easy to

understand for those who have some programming experience. A few things are worth to mention however:

First of all it should be noted that intention of the example isn't to show a script that solves the simplest of dispatching problems. Such a script could certainly have been written with fewer lines of code and higher readability. The script intends to be simple but it also has the goal of solving a task that would have been impossible to solve with a static rule-based approach dispatcher. As for an example the script stores the old speed of the robot when it is stopped to be able to restore it later, this wouldn't have been possible with a rule-based dispatcher since it would lack variables.

The script begins by executing the command `dofile("peis_util.lua")` that loads the PEIS auxiliary library which contains some utility functions for PEIS. Further down the file a callback function named `onrecog` is declared, this function will later be registered as a callback function with the call to `add_callback` at the end of the file. The callback function will not be called immediately however, to enter the callback mode it's necessary to run the function `loop_callback`. This function never returns though, so the script will only be run when a "hooked" tuple changes, if this behavior is not desirable there is another function called `step_callback` that returns but depends on being called periodically.

There are a few things to note about the code that relate to the Lua language itself and its use of tables. The most important (and the only complex) data structure in Lua is the table which is constructed with the `{...}` constructor syntax. In this example there are several examples of the use of tables, what is not visible however is the fact that Lua table indices starts at 1 and not 0 as in C, Java, Python etc. The programmer is not limited to using numerical indices though, as can be seen in the example, assignments to literal indices are also allowed. Lua's small standard library also relies heavily on the use of variadic functions which means functions that take a non predetermined number of arguments. Functions can also return any number of arguments without encapsulating them into a table, if this is done they can either be stored one-by-one with the `a, b, c = foo()` syntax, or packed into a table in the following way: `t = { foo() }`. A table can be unpacked with the `unpack` function which is necessary if one wants send each of the variables in a table as separate arguments to a variadic function.

PEIS specific variables

The following global variables are set automatically in the scripts before they start running:

this_id – Contains the ID of the component running the script, i.e. the scripts PEIS id.

this_key – Contains a base key path that the script could use when setting keys locally, eg. “scripts.testscript”.

PEIS C bindings

These functions are implemented in C and make it possible to access the PEIS environment from within Lua.

<i>add_callback(id, key, fcn)</i>		
Description:		Registers a callback function that will be called when a tuple has changed. The added callback function will be called with the arguments (id, key, val) from the function <code>step_callback()</code> where id and key will contain the real id and key of the tuple that has been changed and val will contain the tuples new value.
Arguments:	id	The id of the component that this callback handles or -1 for all devices.
	key	The key that this callback is registered for, it can contain wildcards.
	fcn	The function that should be called when a tuple has been changed.
Return values:		None

<i>step_callback()</i>		
Description:		<p>Steps the callback system and calls functions registered with <code>add_callback</code> if their corresponding tuples have changed. This function should be called periodically so that internal callback buffers won't get full. If a buffer gets filled up incoming changes will be ignored.</p> <p>This function is intended to be used in scripts that need to perform tasks periodically but still wants to use the callback functionality. E.g.:</p> <pre>while true do i = i + 1; step_callback(); sleep(1); end</pre> <p>If the script doesn't need to run continuously a preferable choice is to use the <code>loop_callback()</code> function defined in the PEIS auxiliary library.</p>
Arguments:		None
Return values:		None

<i>sleep(seconds)</i>		
Description:		Pauses the script for a specified time with the usleep() C function.
Arguments:	seconds	The amount of time that the script should be paused in seconds.
Return values:		None

<i>get_tupleval([id], key)</i>		
Description:		Returns all values of the tuples matching the specified mask.
Arguments:	id	Optional id of the tuple owner. If left unspecified or set to -1 all owners will be considered.
	key	The key of the tuple(s) whose values should be returned, can contain wildcards.
Return values:		All found values in no specific order. Examples: <pre>-- Get the (one and only) value of a fully qualified -- tuple or nil if not found. value = get_tupleval(2353, "kernel.name"); -- Store all tuples from all devices in a specified path -- as a table. valueS = { get_tupleval("kernel.*") };</pre>

<i>set_tupleval([id], key, val)</i>		
Description:		Sets a tuple in a local or remote tuplespace.
Arguments:	id	Optional owner id of the tuple that is being set. If not specified the id of the component running this script is used.
	key	The key of the tuple which is being set, should not contain wildcards.
Return values:		Nothing

<i>locate(key, [val])</i>		
Description:		Locates components with a specified key.
Arguments:	key	The key that should be found.
	val	Optional, if defined it constrains the result to only include the ids of those components that has the desired key and where the key contains the value val.
Return values:		Returns the ids of the components found in no specific order. Example: <pre>-- Get the id of the first found component with the -- name tupleview or nil. tupleview = locate("kernel.name", "tupleview"); -- Get the ids of all components with the name -- tupleview as a table. tupleviewS = { locate("kernel.name", "tupleview") };</pre>

PEIS auxiliary library

The following functions are implemented in a Lua and are intended to simplify some of the common tasks one might encounter when using Lua in PEIS.

<i>waitfor(key,[val])</i>		
Description:		Waits until a component with a specific key and optionally value is seen on the network.
Arguments:	key	The key that the function should wait for.
	val	If specified the key contents should also match val to make the function return.
Return values:		Returns the id of the first found component, this function waits indefinitely.

<i>loop_callback([sleepTime])</i>		
Description:		Steps the callback system continuously.
Arguments:	sleepTime	Optional, specifies how long time the function should sleep between calls to step_callback() if defined. If not defined a default sleep time is used instead.
Return values:		This function never returns.

<i>split_list(str)</i>		
Description:		Extracts substrings from a string and returns each match as a separate entry in a table. Parenthesizes and whitespace characters acts as delimiters and are not matched.
Arguments:	str	The string that should be split
Return values:		A table containing each found substring. E.g. <pre>point = split_list("(1.3 4.7)"); -- point := {1.3, 4.7}</pre>

<i>make_list(...)</i>		
Description:		Creates a string representation of multiple arguments suitable to use as a tuple value.
Arguments:	...	Any number of arguments which should be embedded in the string.
Return values:		The arguments in a string representation. E.g. <pre>str = make_list(1.3, 4.7); -- str := "(1.3 4.7)"</pre>

4. Conclusion

This project has dealt with the task of building a speech recognition system for the PEIS ecology. The final solution to the task is the composition of three separate components handling audio input, speech recognition and component actuation respectively. When put together, these three components form a well functioning speech recognition system that can be configured to accomplish all reasonable speech control tasks within the PEIS ecology. Furthermore, each of the components has been written in a modular and PEIS-compliant way which will allow them to be reused in the future if desired. This is particularly the case with PEIS Lua which presents a novel and generic solution to performing lightweight tasks in the PEIS ecology.

Limitations and problems

The accuracy of the recognition is rather good as long as the microphone is at a suitable distance from the speaker and the room is silent. But when the power of the input signal drops too low the accuracy drops as well and PEIS Julius starts to reject the input, or even worse, misinterpret it. Most of the tests were performed using a low budget microphone that generated a lot of static noise so it's possible that the recognition can be improved with the use of a better microphone. Another way of increasing recognition accuracy is by reducing the static noise with a filter for ALSA (Advanced Linux Sound Architecture).



When creating a new grammar, words are typically taken from the Voxforge wordlist which currently contains almost 15,000 words. These words have been trained by people dictating books, newspapers etc., and all of the words found in this grammar should be usable with Voxforge's acoustic model. Unfortunately some words like "Rotate" are missing, there is however a larger wordlist using the same phonemes that contains roughly 130,000 words which can be used instead. Some of the words found in the larger wordlist can't be used directly though and will therefore require the use of HTK to train or synthesize the missing phoneme sequences in the acoustic model.

Future work

There is some interesting work that could be done in the future to integrate speech recognition with the PEIS ecology more closely.

Currently the user is required to edit the grammar and scripts each time a new device is added to the ecology to be able to control it. It would be a useful enhancement if the system could instead be made to adapt automatically to devices which are present in the ecology at any given time. This could possibly be achieved by using a knowledge base that specifies what kinds of actions are possible to perform on each object in the ecology. A knowledge base could also be used to resolve conflicts about which object is being considered at any given time (e.g. the generic term "Robot" refers to a specific robot if there is only one robot in the ecology).

Tests could also be performed to determine if it's possible to make speech recognition available in the entire environment and not just directly in front of a microphone. This would most likely require the use of a number of microphones listening for sounds simultaneously in different parts of the room.

5. References

1. Saffiotti, A., et al., *The PEIS-Ecology Project: vision and results*. Proc of the IEEE/RSJ Int Conf on Intelligent Robots and Systems (IROS), 2008: p. 2329-2335.
2. Broxvall, M., et al., *PEIS Ecology: Integrating Robots into Smart Environments*. Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA), 2006: p. 212-218.
3. *VoxForge*. Available from: <http://voxforge.org/>.
4. *CMUSphinx: The Carnegie Mellon Sphinx Project*. Available from: <http://cmusphinx.org/>.
5. *Open-Source Large Vocabulary CSR Engine Julius*. Available from: <http://julius.sourceforge.jp/>.
6. *The Programming Language Lua*. Available from: <http://www.lua.org/>.