Version 2.1

Phil Shinn

VUID ToolKit

Contents

[Introduction 3](#_Toc80968605)

[License 3](#_Toc80968606)

[Overview 3](#_Toc80968607)

[Requirements 4](#_Toc80968608)

[Getting Started 5](#_Toc80968609)

[Custom Shapes 10](#_Toc80968610)

[Import Shapes 10](#_Toc80968611)

[Shape Property List 12](#_Toc80968612)

[Editing/Adding Custom Properties 12](#_Toc80968613)

[Custom VTK Objects 15](#_Toc80968614)

[Start 15](#_Toc80968615)

[Prompt State 15](#_Toc80968616)

[Arc (Arrow) 16](#_Toc80968617)

[Grammar State 16](#_Toc80968618)

[Global Behavior 17](#_Toc80968619)

[Computational State 18](#_Toc80968620)

[Decision State 18](#_Toc80968621)

[Stop/Return 19](#_Toc80968622)

[Sub-Dialog State 19](#_Toc80968623)

[Grammar 20](#_Toc80968624)

[List Grammar 20](#_Toc80968625)

[Global Grammar 20](#_Toc80968626)

[Digits Grammar 21](#_Toc80968627)

[Python State 22](#_Toc80968628)

[Init DB 22](#_Toc80968629)

[AccessDB 23](#_Toc80968630)

[Example 1 – Basic Call Flow 24](#_Toc80968631)

[Page 1 24](#_Toc80968632)

[Page 2 25](#_Toc80968633)

[Example 1 Interactions 25](#_Toc80968634)

[Example 2 – Digits Grammar and Dynamic Results 27](#_Toc80968635)

[Example 2 Interactions 28](#_Toc80968636)

[Example 3 – User Developed Python Extensions 29](#_Toc80968637)

[Example 3 Interactions 30](#_Toc80968638)

[Example 4 – Database Extension 31](#_Toc80968639)

[Sharing a Design 32](#_Toc80968640)

[Export to HTML 32](#_Toc80968641)

[Export to/Import from XML 32](#_Toc80968642)

[Cloud Sharing with Draw.io 32](#_Toc80968643)

[Other Tools 34](#_Toc80968644)

[Exporting the XML File 34](#_Toc80968645)

[Running the Toolkit GUI 36](#_Toc80968646)

[Extracting the Prompt List 37](#_Toc80968647)

[Running the Application Live 42](#_Toc80968648)

[Running Simulations & Creating Test Scripts 44](#_Toc80968649)

[To Do 48](#_Toc80968650)

[Upgrade User Interface for Chat 48](#_Toc80968651)

[Port XML loading error messages into GUI 48](#_Toc80968652)

[Integrate with MSFT’s Language Understanding Intelligent Service (LUIS) 48](#_Toc80968653)

[Build Hooks into PySpeech 48](#_Toc80968654)

[Build VXML Code Generator 48](#_Toc80968655)

[Document Understanding 48](#_Toc80968656)

[Question Answering 48](#_Toc80968657)

# Introduction

This document describes the Voice User Interface Designer Toolkit (VUID Toolkit or VTK), a collection of tools that supports designing and maintaining a voice user interface (VUI). The original code was written by Phil Shinn and Matt Shomphe, and placed into the public domain in 2007. Our goal was to make life easier by helping to set up a uniform standard markup language for VUI designs. Since then, a number of organizations have adopted the toolkit for their own use.

# License

The code is licensed under the GNU General Public License version 3 (GNU GPLv3), which is described here: <http://www.gnu.org/licenses/quick-guide-gplv3.html>. These are the terms: “This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details”

Yes, the code is free, as in free beer. If you would like support, contact Phil at [phil.shinn@gmail.com](mailto:phil.shinn@gmail.com) or call +1 (646) 736 -2377.

# Overview

VTK can be used to design an interactive application. You could stop there, or, use some of the simpler ‘helper’ applications in the toolkit to do things like extract the prompt list.

In addition, you can use VTK to develop the suite of test cases associated with the application, so if you change the GUI, it is easy to update the test cases. The test cases are in Cyara xml format, so they can be directly imported into Cyara portal and executed as part of a regression suite. See [www.cyarasolutions.com](http://www.cyarasolutions.com) for an overview of Cyara technology.

This version of the VTK also has an interactive text mode, so you can run the application as a text-based application, based on the design.

Some additional features of this version, compared to VTK 1.0, is this one works with DrawIO, which is a Chrome-based graphical UI. This removes the requirement that a user have a copy of Visio, and in addition, supports both Microsoft Windows and Apple’s OSX operating systems.

This version compared to the last supports Python 3.+

# Requirements

The current version has been tested on Windows 10, but should work on XP and all later versions of Windows. It has also been tested running OSX 10.10 (Mavericks).

Chrome, the Google browser, can be downloaded here: <https://www.google.com/intl/en/chrome/browser/desktop/>

Draw.io Desktop is used for the GUI. To get the desktop version, go to the Chrome web store and download the Draw.io Desktop (<https://chrome.google.com/webstore/detail/drawio-desktop>). This is a free plug-in to Chrome that replaces Visio[[1]](#footnote-1). You can also use the Drawio in the cloud (use www.drawio.com).

Python 3+ available for free download here: <https://www.python.org/downloads/>. . If you have an Apple computer, Python is probably already installed on your machine. See <https://docs.python.org/3/using/mac.html>

# Getting Started

After you’ve got Chrome, and have installed the Draw.io plug-in (see Requirements, above), you can launch Chrome/Draw.io via the Google Chrome App Launcher:

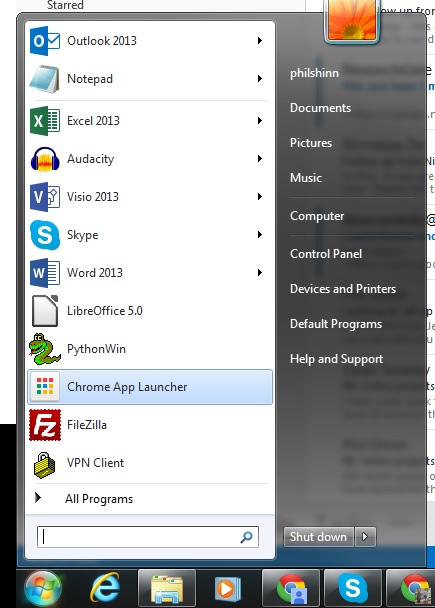


Figure Chrome App Launcher

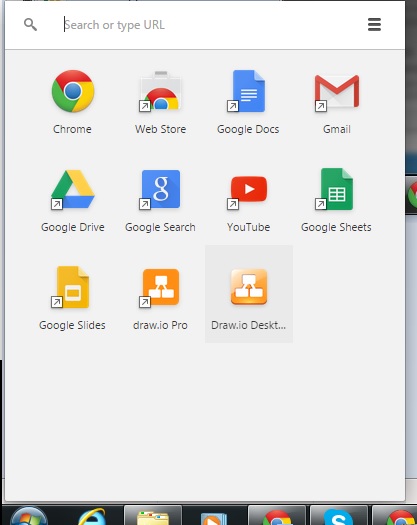


Figure 2 Selecting Draw.io Plugin

Select Draw.io Desktop and launch it:

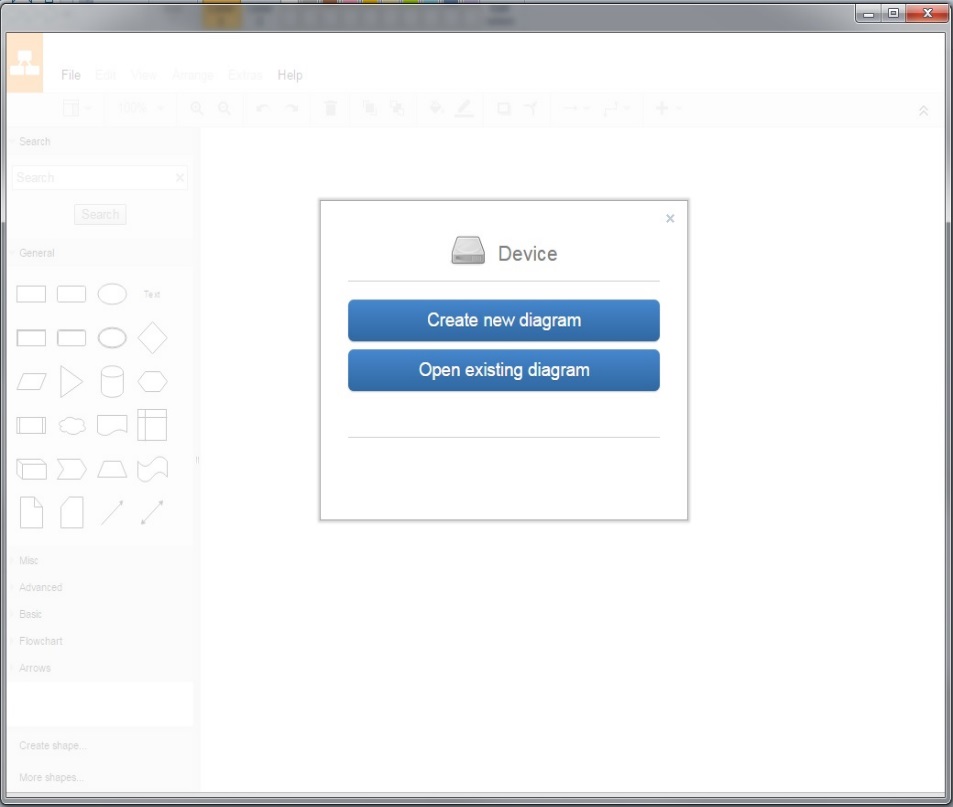


Figure Draw.io desktop Launched

Drag the file VTK 2.33.xml into the Draw.io canvas:

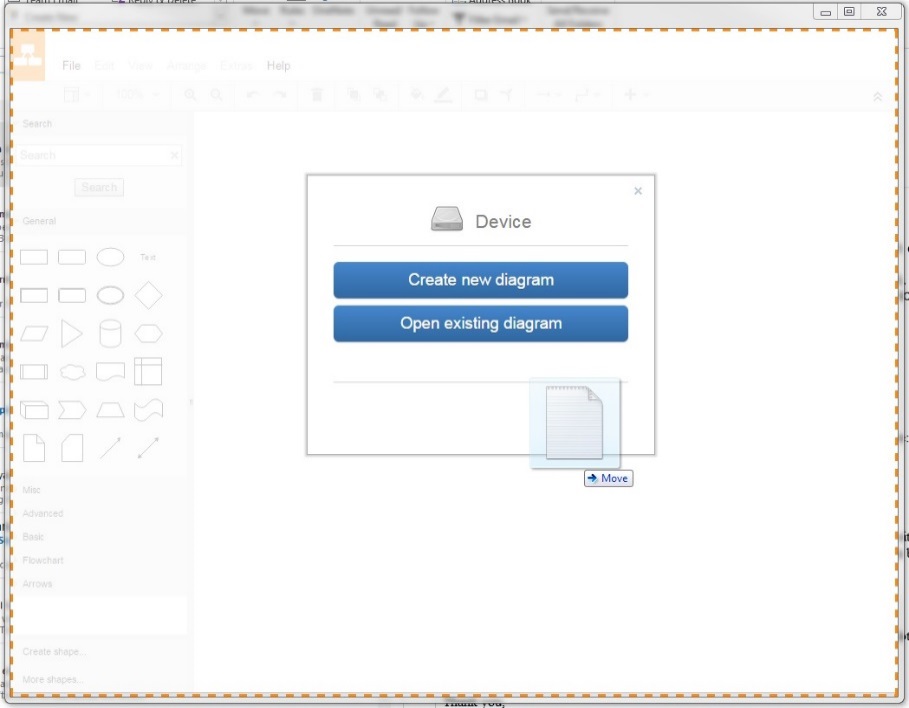


Figure Drag xml into Draw.io

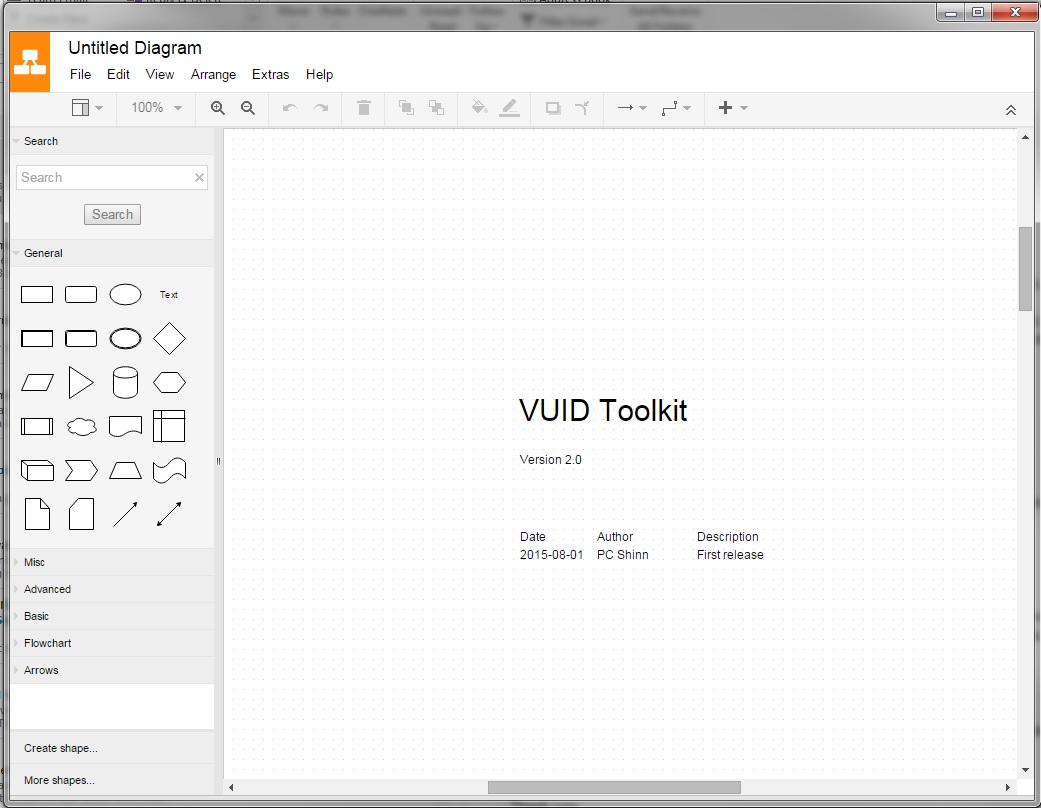


Figure Ready to go!

And you’re ready to start designing.

You might want to save your new design with a new name and not change the original VTK .xml file, since if you break it to start over with a new design you’ll have to download it again.

The first page, or tab, in the template document is used as a cover page with a version number and a list of changes. The second page is a legend that describes the custom objects used by VTK. Take a moment to read this page. The rest of the pages contain a few sample applications.

# Custom Shapes

A VTK application is a collection of custom shapes that are used to build a [directed graph](https://en.wikipedia.org/wiki/Directed_graph). More pedantically, you build a [rooted directed graph](https://en.wikipedia.org/wiki/Rooted_graph), because you can tell the application where to start. Since there is also a way to specify where a graph ends, you can group different graphs into modules and hook them up to build sub-routines. More on this later.

There are (currently) 15 custom shapes in the toolkit. These shapes are sufficient to design an application, and execute it, but you can create/extend/modify other shapes if you want.

## Import Shapes

You can use the shapes from the legend page of the example xml file, but you can also load the shapes into the DrawIO shapes panel from an external library file. Then when you are deep into a design and you need a new shape, you don’t need to skip back to the legend to get the shape, you can just grab it from the shapes panel:

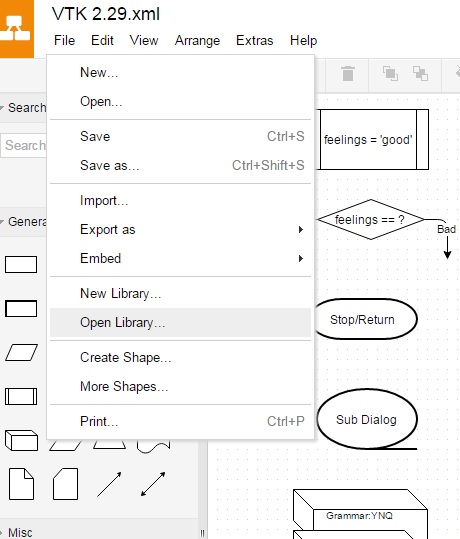


Figure : Load Customer Shapes from Library File

Choose the shape library file named VTK\_Library.xml:

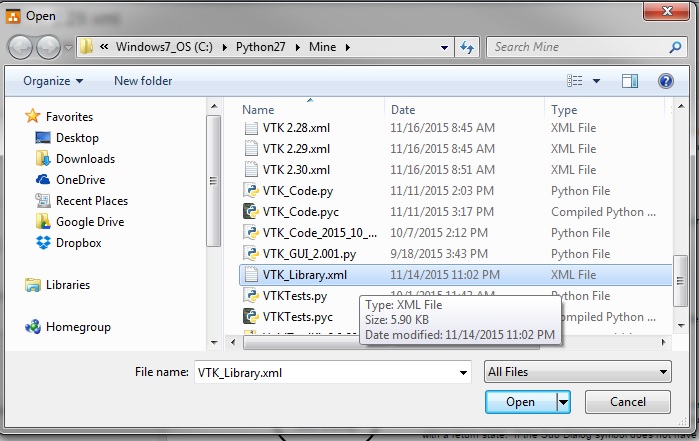


Figure : Select VTK\_Library.xml

The shapes are now available for drag and drop onto the design.

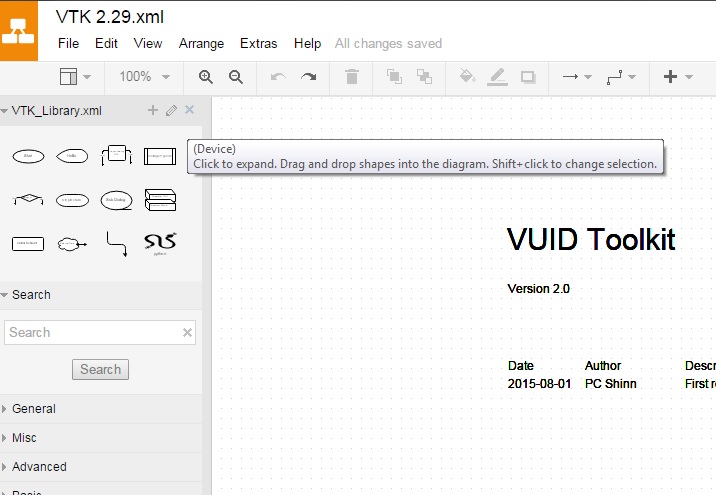


Figure : VTK\_Library loaded into shapes panel

You can add new shapes or groups of shapes into the library. Create the shape you want, select it, then click on the ‘+’ symbol in the library panel:

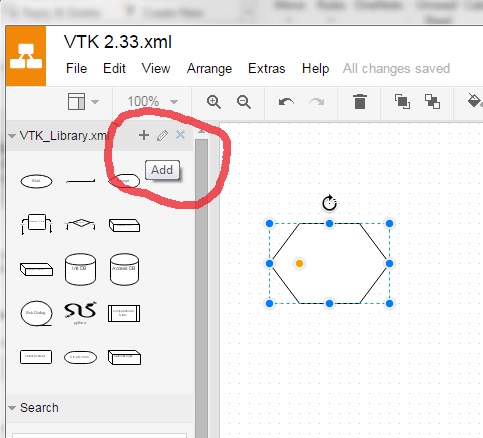


Figure : Adding custom objects to library.

## Shape Property List

Each shape has a property list associated with it. The property list is a list of key-value pairs. To see an object’s property list, in the design hover over it with your mouse (see the yellow boxes below).

# Editing/Adding Custom Properties

To edit the property list of an object, or to add your own custom properties to the existing objects, select the object, and then from the Edit menu select Edit metadata… :

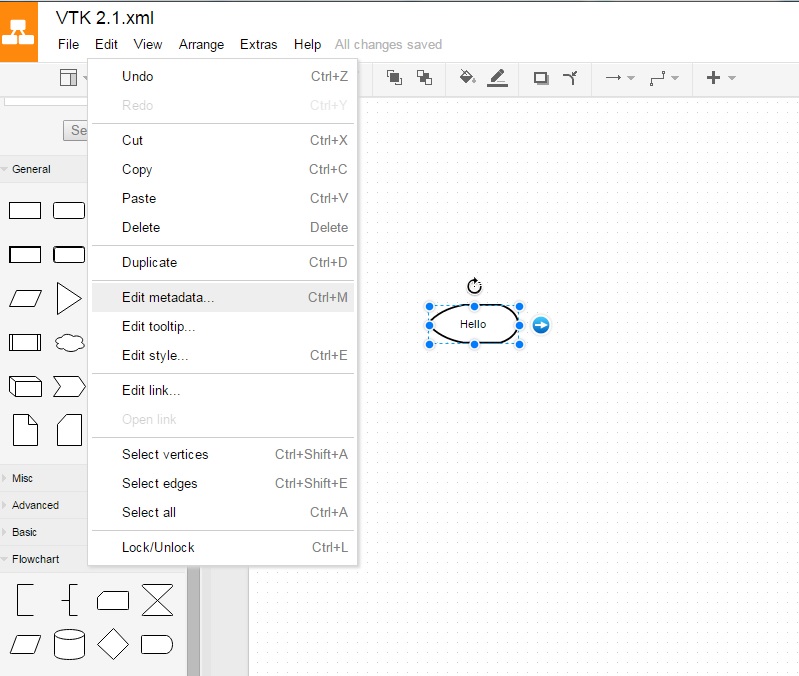


Figure : Editing object property list

That brings up the property list:

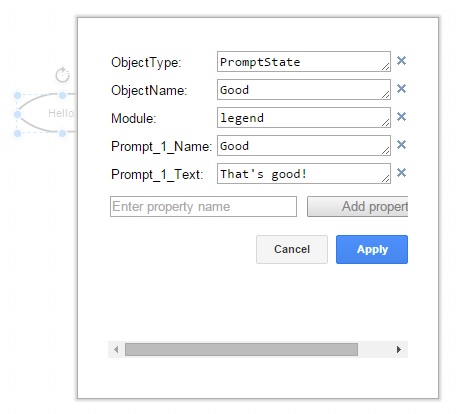


Figure : Object Property list.

To add an additional property, in this case Language, fill in a new one in the Enter property name field and select Add property then type in the value. Finish by selecting Apply.

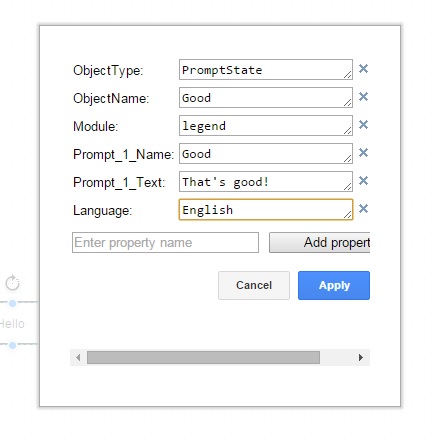


Figure : Adding new property.

# Custom VTK Objects

The existing keys and values in the example xml files are used by other parts of the system. You should not change the keys of the existing objects. But feel free to add new properties and values. You should also not change the values of the property ObjectType.

Here is a brief summary of the custom VTK objects.

## Start

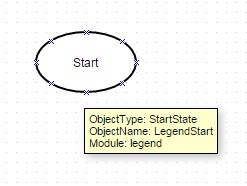
For this object, the property ObjectType has the value StartState. You should not change this property or its value. The property ObjectName has the value LegendStart, and you can (and should) change this value depending on the module you are working on. It is best practice to give each unique object a unique ObjectName. The property Module refers to a logical collection of nodes, for example a credit card number collection. There is one start state to a module, and only one outgoing arc from a Start State is allowed. Make sure to change the module property from ‘legend’ to whatever new module you are creating.

Figure 13: Start State

## Prompt State

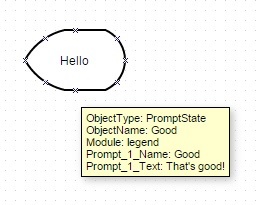
Use a prompt state to specify what the system says to the user. There can be one or more input arcs, but only one outgoing arc allowed. The ObjectType of PromptState should not be changed. This prompt state has one prompt, whose name is Good and whose text is That’s good!. You can add as many as you want in the properties list, using the format Prompt\_N\_Name - Prompt\_N\_Text where the prompts are rendered in ascending N order.

Figure 14: Prompt State

## 

## Arc (Arrow)

If there is only one arc exiting a state, it is unlabeled. For grammar states or decision states (see below) the label on the arc represents either what the user said or what the value is of a variable.

Figure 15: Arc (Arrow)

## Grammar State

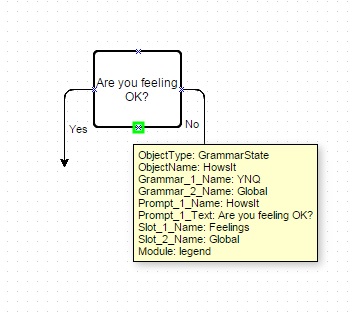
A grammar state is used to collect input from a user. It is typically designed to output prompts before it does the collection of the user’s input and parses it with a grammar (see below).

Figure 16: Grammar State

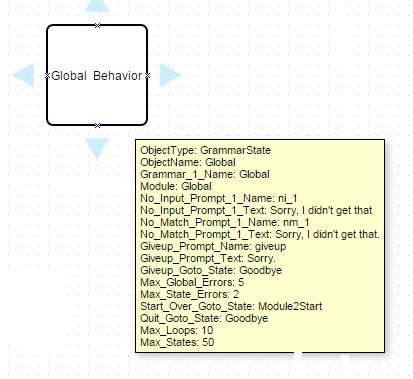
Labels on the outgoing arcs represent where to go next in the application based on the user's semantic intent. Prompts follow the same conventions as discussed in the prompt state above. Add as many grammars as required.

The property list has 2, a state-specific grammar (e.g., YNQ) and a global grammar. Follow the same convention as prompts for additional grammars.

Create a grammar object (see below for examples) for every grammar mentioned in the properties list. The slot name in the property list is the field name in a grammar which is populated with a value in the user's response.

## Global Behavior

Figure 17: Global Behavior



The global behavior is a special kind of grammar state that must be included in a diagram.

This is where the no input and no match error handling prompts are specified.

A no input prompt is what the system would say after a certain amount of time if the user does not reply to a question.

A no match prompt is what the system will say if the user replies with something the system does not understand.

The Giveup prompt is what the system says when it is giving up, and the Giveup\_Goto\_State is where it goes after Max\_State\_Errors are experienced.

## Computational State

Figure 18: Computational State

Use a computational state to set variables to values. In this example case, the variable named feelings is being set to the value good. These variables and values can be checked by other states, such as a Decision State (see below).

## Decision State

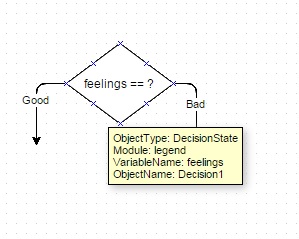
Used to make decisions about where to proceed depending upon the value of the variable. In this case, if the variable named feelings has the value good, then follow the path to the left. If the value is bad, then follow the path to the right.

Figure 19: Decision State

## 

## Stop/Return

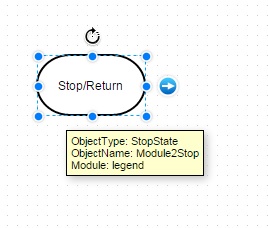
Used to end a module or the whole parent application. If the stop state is inside a sub dialog (see below), and there is an arc coming out of the sub dialog symbol, then flow continues back to the state pointed at by the parent sub dialog symbol. If there is no arc coming out of the parent sub dialog symbol, the application stops.

Figure 20: Stop/Return State

## Sub-Dialog State

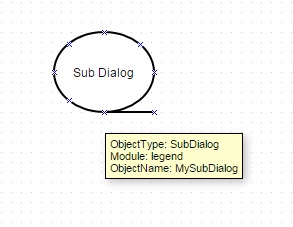
Refers to a separate graph usually existing in another module on a different page. It should have a start state and optionally a return state. By convention, if a Sub Dialog has an exit arc, it acts like a subroutine with a return state. If the Sub Dialog symbol does not have an arc that exits the symbol, than treat it as a labelled Goto, meaning flow of control does not return.

Figure 21: Sub-Dialog State

## Grammar

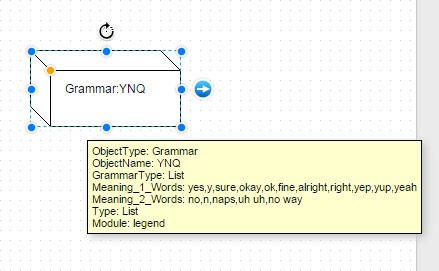
A grammar is used in a Grammar State to parse a user’s input.

Figure 22: List Grammar

In this example from the legend, there are two meanings. The first item in the list of words in a meaning is also used as the general semantic intent associated with the meaning.

So in this example, the first meaning is yes and the words associated with that meaning are yes, y, sure, okay, fine, etc. The second meaning is no.

### List Grammar

Every grammar needs to have a property name Type, and in this example, the type of the grammar is List. When this grammar is applied to a user’s input, if the input is yep, then the semantic intent returned to the application by the grammar is yes.

### Global Grammar

In addition to the Global Behavior item discussed above, there can also be a grammar named Global, which if present, has special treatment in that if there are two grammars at a state and one of them is named Global, and both grammars have the same terms (i.e., the response from the user is ambiguous) then the default application behavior is to follow the global grammar. You can change this is you want.

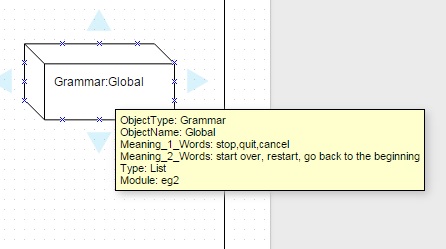


Figure : Global Grammar

The global grammar has to be referenced in each grammar state in order to be used. In other words, if the global grammar is not in the list of grammars active at a given grammar state, then those words won’t be understood at that state.

### Digits Grammar

A second grammar type in the demo package is Digits. An example of this being used is given below.

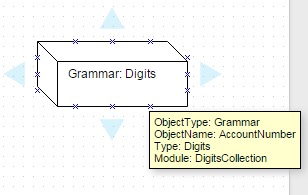


Figure : Digits Grammar

## Python State

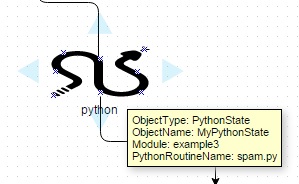


Figure : Python State

The python state is used to call a user-defined python script. This is useful when you need to do general purpose computing that requires more functionality than the simple variable setting done in the computational state. For example, suppose you have an application that has vocabulary like “Turn off the lights” or “send an email to Mary.” If you wanted to have the application actually go out and do those things, you could use the python state to implement turning off the lights or composing an email.

In the example set it is shown being used in the module example3. The key named PythonRoutineName holds the name of the python script (spam.py, which is in the base directory) that will be called by the application when that python state is run. There can also be a return value returned to the state that can be used if desired.

## Init DB

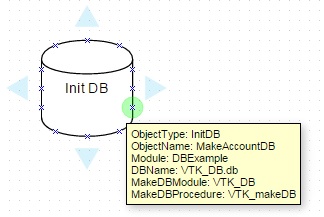
The InitDB object is used to create a database. In the example application xml file, in the module DBExample, there is an InitDB object that refers to a SQLite database that implements a basic accounting database. The Module key refers to the state machine module that the InitDB object belongs to. The name of the database is VTK\_DB.db, and is in a file in the base directory.

Figure 26: InitDB

The key MakeDBModule refers to the name of the python script file that is used to implement the creation of the database. In this example, the name of the python script file that is loaded, and which exists in the base directory, is VTK\_DB.py. Note that you refer to it in the GUI without the .py extension. Any InitDB objects that are included in a design are initialized when a state machine is created, so notice that this one is not attached in the flow.

The key MakeDBProcedure refers to the name of the function in the python script file that is executed to create the database. In the example, that is VTK\_makeDB. So to see what is populated in the database, you can open up the file VTK\_DB.py and look at the function named VTK\_makeDB.

## AccessDB

The AccessDB object is used at run-time to access the database. The Module key refers to the state machine module that the InitDB object belongs to. The name of the database is VTK\_DB.db, and is in a file in the base directory. An example of use is given below in DBExample.

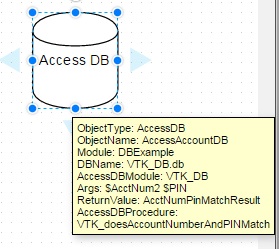
DBName is the name of the database to use. The key AccessDBModule refers to the name of the python script file that contains the functions that access the database. In this example, the name of the python script file that is loaded, and which exists in the base directory, is VTK\_DB.py. Note that you refer to it in the GUI without the .py extension.

Figure 27: AccessDB

The key AccessDBProcedure refers to the name of the function in the python script file that is executed to access the database. In the example, that is VTK\_doesAccountNumberAndPINMatch.

If you look at that function in the file AccessDBModule.py, notice the SQL statement that is executed against the database

The key Args gives the arguments that are passed into the access procedure, in this case, the variables $AcctNum2 and $PIN. ReturnValue is the name of the variable where the return value from the procedure is stored. This variable name is created and the return value is stored in the run-time database. It will contain either True or False.

# Example 1 – Basic Call Flow

Here is an example design that shows how to use the elements of a basic call flow.

## Page 1

This application starts by asking the user if they are OK, then it sets the variable ‘feelings’ to whatever

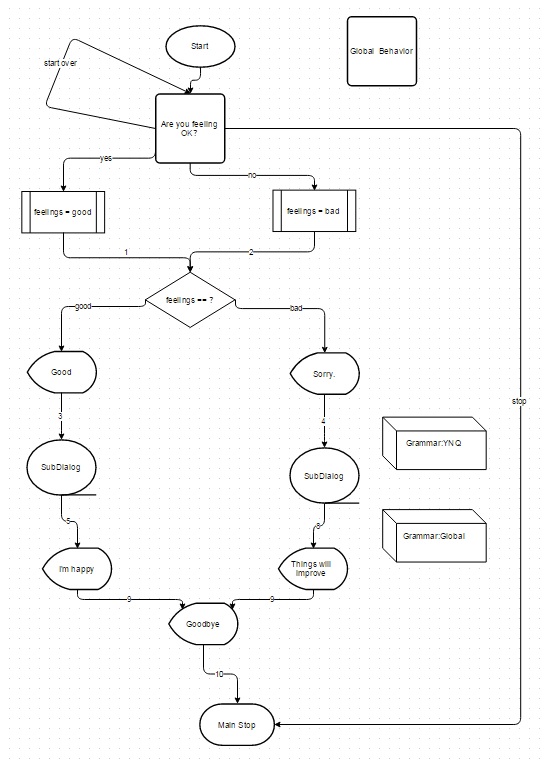


Figure : Example 1, page 1

the user responded with. Next, it branches at a decision node on the value of that variable, and responds appropriately. Control passes on to a SubDialog named, appropriately, SubDialog, which is shown below, and then returns to say goodbye. Also seen is the global behavior of the phrases ‘stop’ or ‘start over.’

## Page 2

This is the SubDialog, which just asks the user their favorite color.

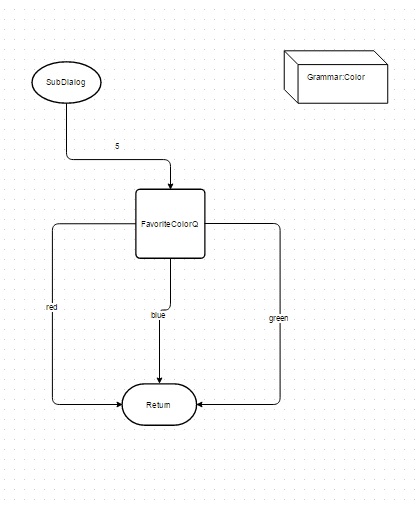


Figure : Example 1, page 2.

### Example 1 Interactions

Example interactions starting with the first application are as follows:

System: Are you feeling OK?

User: Yes

System: That’s good! What’s your favorite color?

User: Blue

System: Goodbye.

System: Are you feeling OK?

User: No

System: Sorry to hear that. What’s your favorite color?

User: Blue

System: Goodbye.

System: Are you feeling OK?

User: Start over.

System: Are you feeling OK?

User: Quit

# Example 2 – Digits Grammar and Dynamic Results

This next small application demonstrates the use of a special type of grammar, a digits grammar, and how to use the result dynamically.

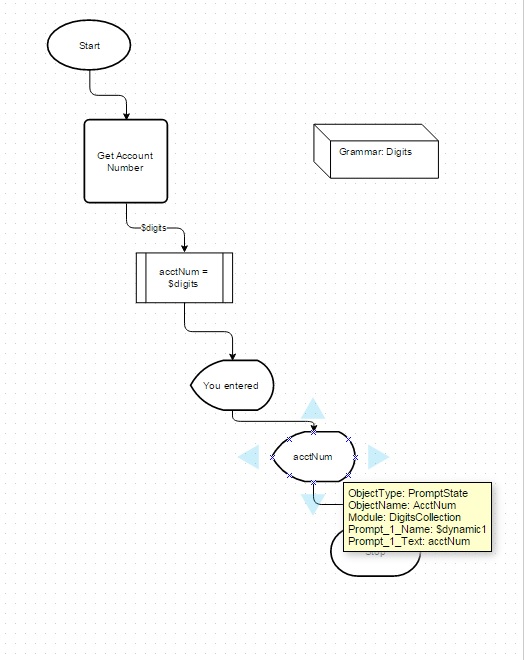


Figure : Example 2

When the Get Account Number grammar state executes, it prompts the user for digits. The single arc $digits represents what the caller types. In the next computational state, whatever has been collected into $digits is assigned to the variable named AcctNum. Later on, in the prompt state AcctNum, the prompt\_1\_Name has a special value $dynamic1, which indicates to the program to take the contents of the variable named acctNum and play that back.

### Example 2 Interactions

Example interactions starting with the second application are as follows:

System: What’s your account number?

User: 1234

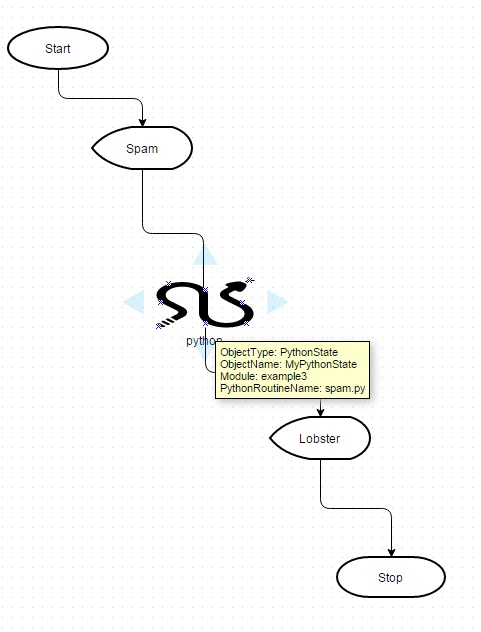
System: You entered 1234

System: What’s your account number?

User: stop

System: goodbye

# Example 3 – User Developed Python Extensions

Figure 31: Example 3 Flow

The third example shows the use of the python state. This application renders a prompt, then calls the program spam.py. Spam.py simply prints “Hello world!” to the console, and returns the string ‘aaa’ to the application.

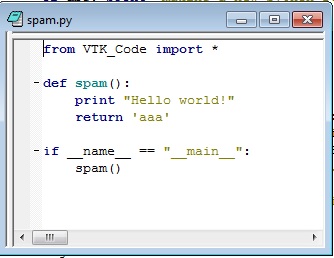


Figure : Spam.py

### Example 3 Interactions

There is no interaction in this example, but the system does write “Hello World” into the console from the referred to python script.

System: Egg, bacon, sausage and Spam Lobster Thermidor aux crevettes with a Mornay sauce, garnished with truffle pâté, brandy and a fried egg on top, and Spam.

# Example 4 – Database Extension

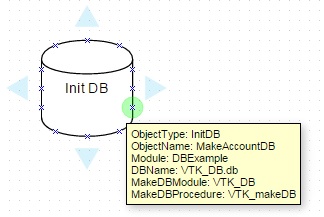
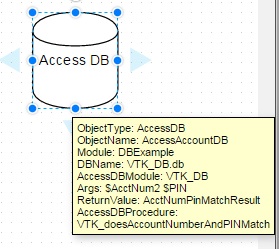
In this example, when the state machine is first created, it builds out a database as described in the InitDB object. Open up the file VTK\_DB.py and look at the function VTK\_makeDB to see the database’s table structure and initial data values.

Figure 33: Example 4 - Using a Database

This flow uses the first parts of the previous example to collect an account number and PIN. The collected values are stored in the run-time database in the variables named AcctNum2 and PIN.

After the app collects these values, it runs the Access DB object:



When run, this object executes the function VTK\_doesAccountNumberAndPINMatch (which lives in VTK\_DB.py). The return value of that lookup is stored in the run-time database in the variable named AcctNumPINMatchResult. That value is then looked up in the decision state, which branches the flow to the appropriate prompt.

# Sharing a Design

There are a couple of options for sharing a design.

## Export to HTML

From the main menu select File > Export as > HTML. This method exports a picture of the design which cannot be further manipulated within the design tool. But it is a good method for distributing the design for others to review, since all they need to view it is a browser.

## Export to/Import from XML

From the main menu select File > Export as > XML). Importantly, uncheck all the boxes, particularly the box labelled Compressed. Use this method to export the drawing if you want to share it with others who want to manipulate the data in the file. This is the method to use to create a starting file for the rest of VTK. There is another option File > Export as > XML (Compressed) which Draw.io uses to save files in native format. You can use this to share designs with others using Draw.io, but the compressed version is not readable by the other tools in VTK. You can then email the exported xml file to a colleague who can import the exported file in Draw.io using File > Import From.

## Cloud Sharing with Draw.io

There are two browser-based cloud versions of Draw.io. One is a single user version at <https://www.draw.io/> which provides the same utility as the desktop version, except that the logic executes in the browser, so no plugin is required. Storage options including Dropbox, Google Drive, OneDrive, the Browser itself or local device.

The other version supports on-line real-time collaboration among multiple users. This version is available at <https://drive.draw.io/> is also known as draw.io Pro which you can access here: <https://chrome.google.com/webstore/detail/drawio-pro/onlkggianjhjenigcpigpjehhpplldkc?hl=en>

Here is a description from the Google Chrome webstore:

**Diagrams Online - The most tightly Google Drive integrated diagramming application available.**

draw.io pro is a diagram editor, built around Google Drive (TM), the technology of which is used in 74 of fortune 100 companies.

draw.io Pro is permanently free for all personal Google accounts.

draw.io focuses on providing enterprise grade:

- Scalability

- Reliability

- Security

- Availability

- Privacy

The key advantages of draw.io for a company basing their cloud storage strategy on Google Drive are:

- All application data is stored only in Google Drive.

- Diagram model is only transmitted directly between browser and Google Drive, server-side data is subject to Google security only.

- draw.io only operates from the https domain, using industry standard SSL encryption.

- Multiple Google accounts are supported, including any number of Google for Work accounts.

- draw.io is hosted on Google App Engine and statically serves a fully operational JavaScript application. The scalability and reliability of draw.io correlate with that of the serving infrastructure only (100% uptime in 2013).

- Your existing backup of Google Drive files will include all draw.io files.

There is some useful information and tutorial videos here on the support page: <https://support.draw.io/display/SUP/draw.io+Online+Support> . These are also hosted on YouTube here: <https://www.youtube.com/watch?v=PDegJXSB2Ho> and here: <https://www.youtube.com/watch?v=TyDqWtNYyrA>

The second tutorial shows how the draw.io.pro version supports real-time simultaneous collaboration, along with chat, and discusses in detail how to share data across the different versions. The on-line version’s manual is here: <https://support.draw.io/display/DO/Draw.io+Online+User+Manual>

One point to take into consideration is they state that these technologies will be free for all existing users from now on for the life of the technologies. So if they do make a decision to start charging for the hosted bits sometime in the future, if you are an existing user at that time they say they will not charge you. Of course if you have downloaded the plug in desktop version then you have it as long as you have the browser and plug in.

# Other Tools

The next set of actions operate on the exported XML file. Things you can do include:

* Extracting the prompt List
* Extracting the list of grammars
* Experimenting with live text interaction
* Generating test scripts
* Generating user journeys

To run the graphical user interface of VTK, you will need to have Python installed. See <https://www.python.org/downloads/> .

## Exporting the XML File

The tools run on an exported XML file of the graphical design. To export the XML file, choose File -> Export as -> XML.

Graphical user interface, application, table

Description automatically generated

Figure : Export XML (Plain!).

Then***, very important!*** uncheck the option box labelled ‘Compressed’. If you don’t the other python routines won’t work, although you could use compressed to share a design.

Graphical user interface, application

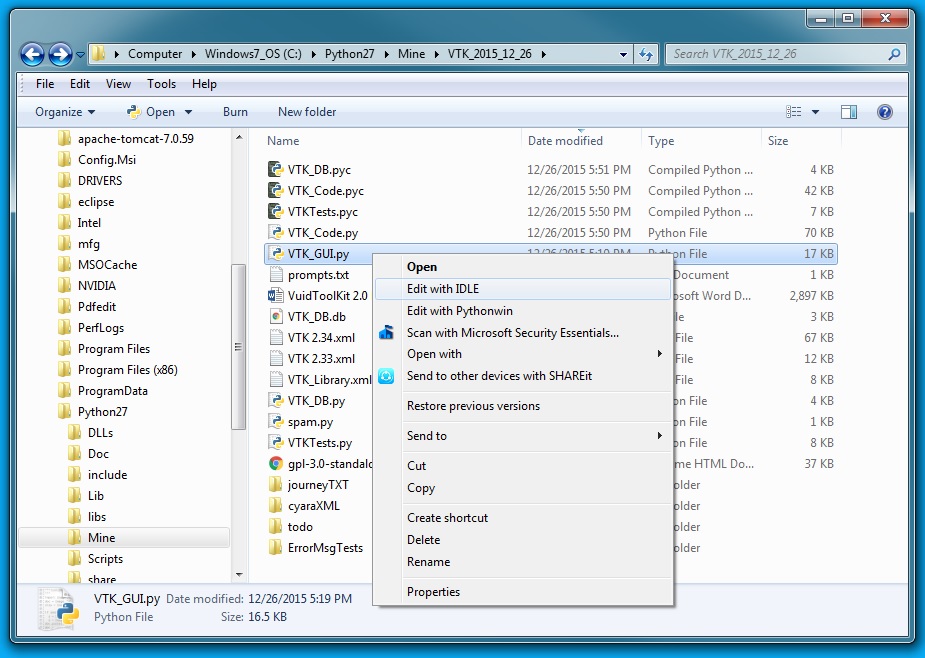
Description automatically generated

Figure 35 Export - Uncheck Compressed

You can view the exported XML with a browser like Chrome or Edge, or use apps like Atom.

## Running the Toolkit GUI

When you have python installed, you can run the GUI directly by double clicking on VTK\_GUI.py, or you can right-click on the file VTK\_GUI.py and choose Edit with IDLE:



This brings up the Python IDE (Integrated Development Environment), and you can then run the functions by selecting Run Module from the IDE menu. In the examples below we show the first method.

After launching VTK\_GUI.py directly, two windows appear – the GUI and a Python console. The console is used for live text interaction.

# Extracting the Prompt List

Load the exported XML file by browsing to and selecting it:

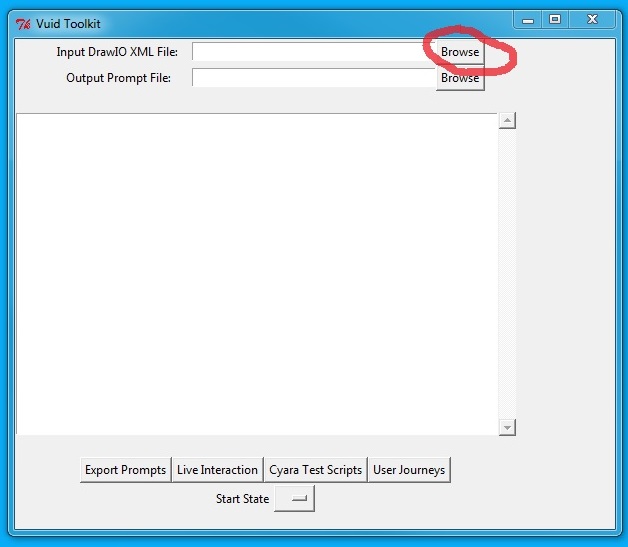


Figure 36 Browse to load xml file

Choose the uncompressed xml file:

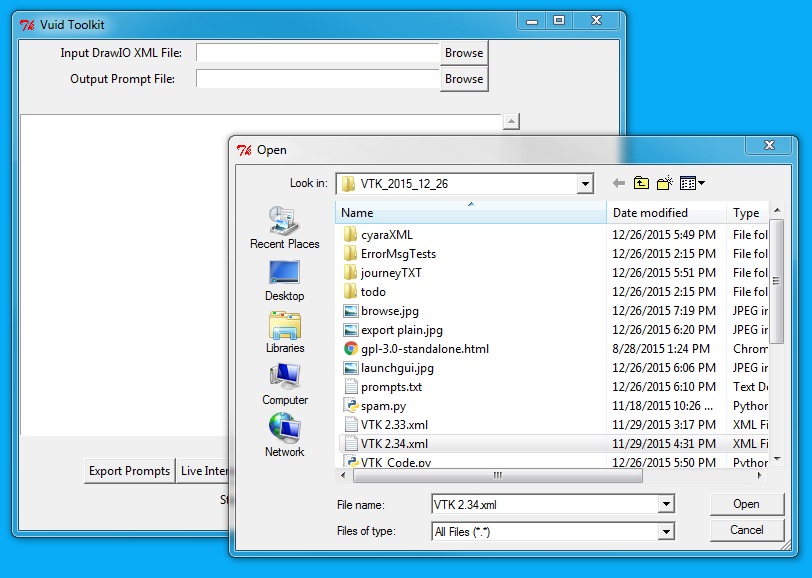


Figure : Choose uncompressed xml file

Next, specify where to write the output prompt file to:

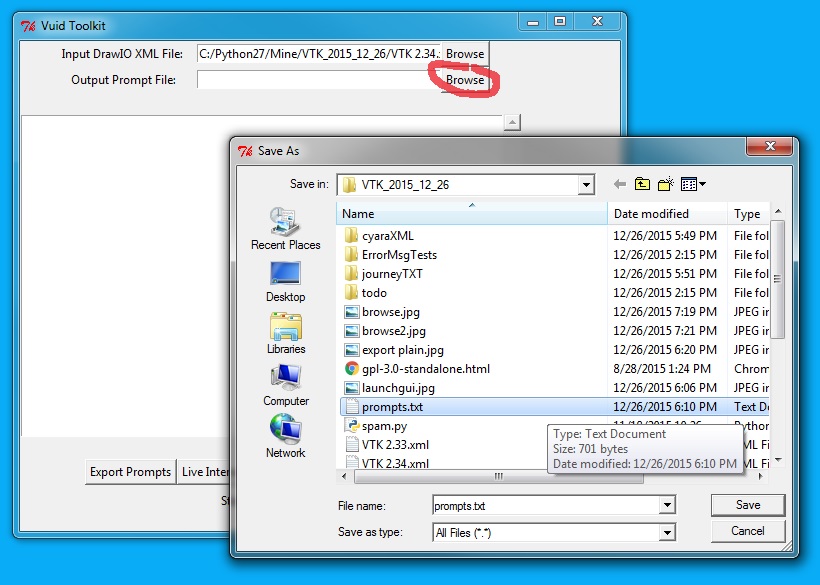


Figure : Specify output Prompt file.

Export the prompt list:

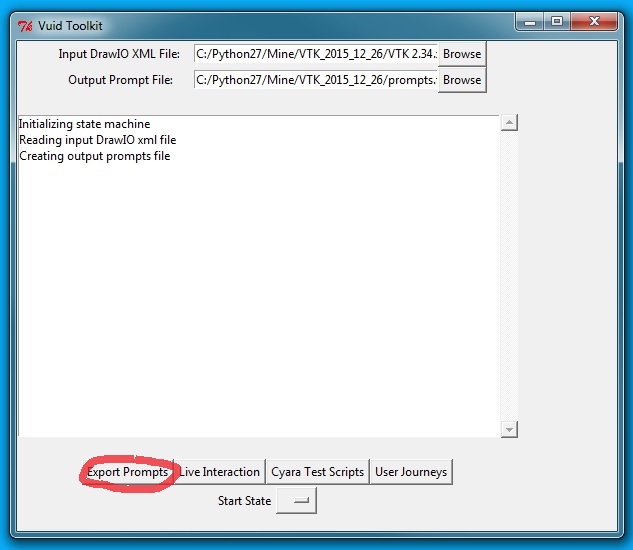


Figure : Export Prompts

The output prompts file is tab-separated prompt file name and text:

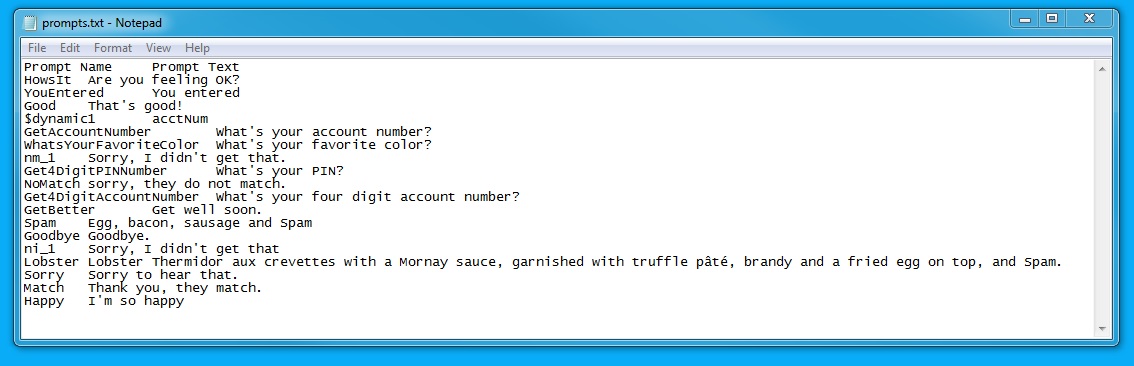


Figure : Ouput prompt file.

Output is written in utf-8 (<https://en.wikipedia.org/wiki/UTF-8>), so you can write your prompts in languages other than English.

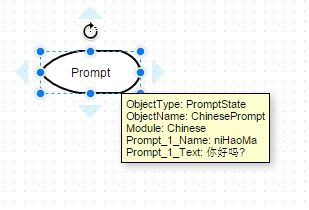


Figure : UTF-8 Supported character set.

(I have yet to work on Chinese input on the matching grammar side, but I don’t see why not…)

# Running the Application Live

To run the application in text input mode, specify the input Draw IO xml file and select Live Interaction. This loads the state machine and collects the list of possible states to choose from:

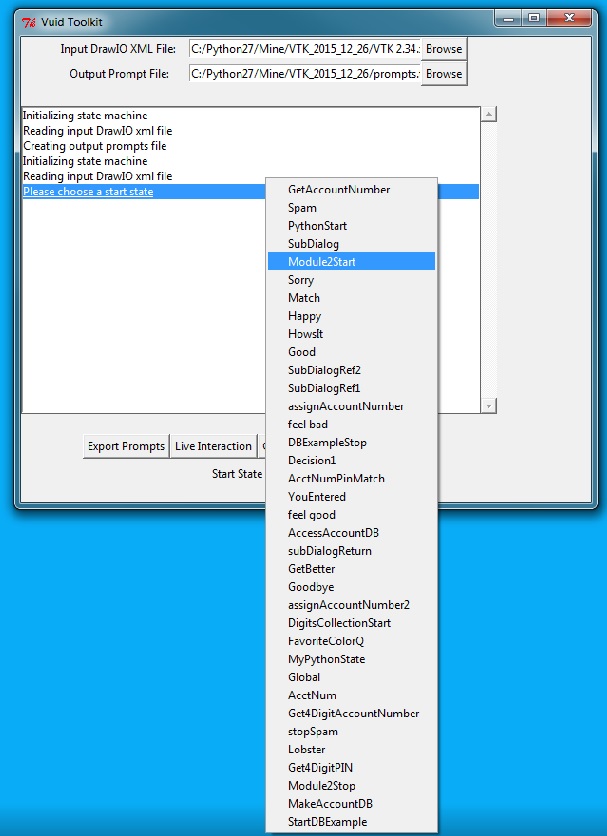
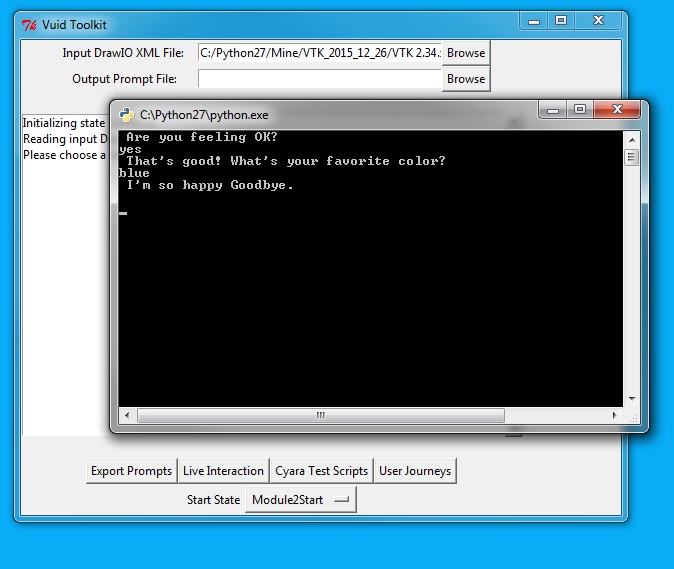


Figure : Running interactive - select start state.

After you specify the state machine to run and the start state, then select the Live Text Interaction button again. The text interaction runs in the python console:

Figure 43: Live text interaction in console.



# Running Simulations & Creating Test Scripts

You can run automatic simulations of your application and create test regression test scripts by using the program’s simulator options. When the simulator runs, instead of taking input from the user live, the system itself creates a range of possible inputs for each grammar state. It then executes the state machine starting at a given start node using all possible combinations of inputs. The results can be written to two formats:

* Journey TXT format is just one text file per path through the application. Use these to do ‘table reads’ of the journeys when in VUI design mode.
* Cyara XML is a format which can be imported into the Cyara IVR testing system (see [www.cyarasolutions.com](http://www.cyarasolutions.com)). When you have completed a design and want to do a load test or run a regression test, with slight tweaking (entering a phone number) the scripts can be used in a live test with the Cyara system automatically generating the phone calls and validating the system performance.

To create the user input for the simulations, add the property Simulations to a grammar state and enter values as follows:

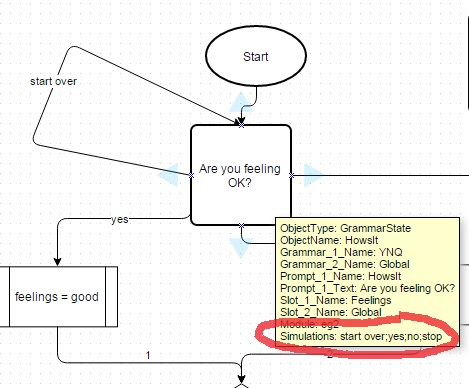


Figure : Simulations values for Howsit grammar state.

When the simulator is executed, four possible inputs for this grammar state will be created: start over, yes, no and stop. In this application, there is another grammar, and in the Simulations property, there are three values listed, red, blue and green:

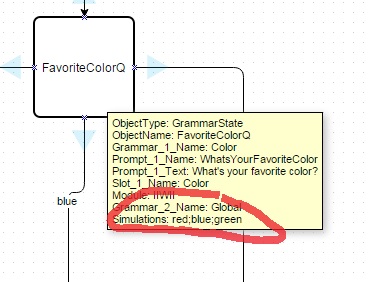


Figure : Simulations values for FavoriteColorQ grammar state.

When the simulator runs, it creates a series of new state machines and executes them using all possible sets of inputs. In this case, there are 4 possible inputs for the first grammar and 3 possible inputs for the second grammar, so the simulator will create and execute 12 possible simulations, one for each possible pairing on inputs (e.g., start over, red; start over, blue; start over, green; yes, red; yes, blue; etc.).

To run simulations and create the caller journey files, specify the input DrawIO file and the start state, then select User Journeys.

The output files are written to a subdirectory named JourneyTXT. Here’s an example of one of the files:

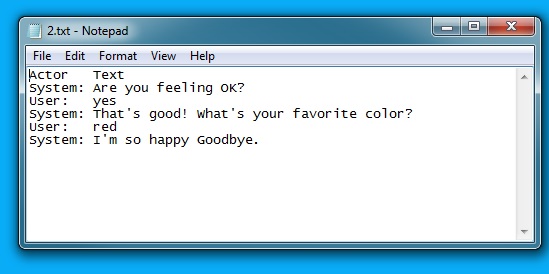


Figure : Example Caller Journey file.

To create the Cyara XML test scripts, specify an input DrawIO xml file and a start state, and then select Cyara Test Scripts. The individual files are written to a subfolder named cyaraXML. Here is a screen shot of a piece of one of the xml files:



Figure : Cyara XML for test case.

# To Do

This is a list of possible features/functionality to implement. In no particular order.

## Upgrade User Interface for Chat

Currently the interaction between user and the system runs from the python IDE. This should be upgraded to put in inside the GUI. In addition, it could be rendered as a web service, allowing anyone to interact who has a browser.

## Port XML loading error messages into GUI

Currently error messages that print when loading the xml only show up in the python IDE. These should be ported to the GGGUUUI.

## Integrate with MSFT’s Language Understanding Intelligent Service (LUIS)

See <https://www.luis.ai/>

## Build Hooks into PySpeech

<https://github.com/michaelgundlach/pyspeech>

## Build VXML Code Generator

Develop code to submit to folks like <http://www.plumvoice.com/>

## Document Understanding

I know this is out there, but build something that reads a website, using something like scrapinghub and make a question answering app.

## Question Answering

Integration with <http://quepy.machinalis.com/> , integration with Wolfram alpha language

1. Currently, the list price of Visio Professional is $590, Visio Standard is $300 and monthly rental is $156 for a year. [↑](#footnote-ref-1)