

# TuTalk Dialogue System Design Specification

*Revision : 1.7*

Brian (Moses) Hall, Pamela Jordan and Michael Ringenberg  
University of Pittsburgh  
Learning Research Development Center  
3939 O'Hara St.  
Pittsburgh PA 15260

March 21, 2012

# Contents

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	Basic Requirements . . . . .	3
1.2	Getting the Files for Local Users . . . . .	3
1.3	Run <code>setup.py</code> . . . . .	4
1.4	Optionally Setting up TuTalk as an HTTP Server . . . . .	4
1.5	Running TuTalk from the Commandline . . . . .	4
<b>2</b>	<b>Architecture</b>	<b>5</b>
<b>3</b>	<b>Dialogue Scenarios</b>	<b>5</b>
3.1	The <code>sc</code> Scenario Compiler . . . . .	6
3.2	Concepts . . . . .	7
3.2.1	Concepts in <code>sc</code> . . . . .	10
3.3	Recipes . . . . .	11
3.3.1	Recipes in <code>sc</code> . . . . .	20
3.4	Transitions . . . . .	22
3.4.1	Transitions in <code>sc</code> . . . . .	25
3.5	Module Configuration . . . . .	25
3.5.1	Configuration in <code>sc</code> . . . . .	25
3.6	The Syntax of <code>sc</code> . . . . .	26
<b>4</b>	<b>Dialogue Policies</b>	<b>28</b>
<b>5</b>	<b>The Coordinator</b>	<b>28</b>
5.1	Usage . . . . .	29
5.2	Command-Line Switches . . . . .	29
5.3	TuTalk Commands . . . . .	29
5.4	Simulating a Student . . . . .	30
<b>6</b>	<b>The XML Communication Format: TuTalkMessage</b>	<b>30</b>
<b>7</b>	<b>tutalkd</b>	<b>31</b>
7.1	The <code>register</code> command . . . . .	33
7.2	The <code>send</code> command . . . . .	33
7.3	The <code>login</code> command . . . . .	33
7.4	The <code>submit</code> command . . . . .	34
7.5	The <code>time</code> command . . . . .	34
7.6	The <code>auth</code> command . . . . .	35
7.7	The <code>privileged-login</code> command . . . . .	35
7.8	The <code>scenarios</code> command . . . . .	36
7.9	The <code>upload</code> command . . . . .	37
7.10	The <code>launch</code> command . . . . .	37
7.11	The <code>coordinator</code> command . . . . .	37
7.12	The <code>account</code> command . . . . .	38

<b>8</b>	<b>Writing a Client</b>	<b>38</b>
8.1	Messages Sent by the Client	39
8.1.1	login	39
8.1.2	input	40
8.1.3	logout	40
8.2	Messages Sent to the Client	40
8.2.1	login-response	40
8.2.2	gid-assignment	41
8.2.3	tutor-turn	41
8.2.4	tutor-logout	41
8.3	Detecting Errors	42
<b>9</b>	<b>Modules</b>	<b>42</b>
9.1	Extending TuTalk	42
9.2	Input	42
9.2.1	Configuration	43
9.3	Normalizer	43
9.3.1	Configuration	43
9.4	NLU	44
9.4.1	Submodules	44
9.4.2	The <code>nlu</code> Query	44
9.4.3	NLU Wizards	45
9.4.4	Configuration	46
9.5	NLG	46
9.5.1	The <code>nlg</code> Service	46
9.5.2	The <code>nlg-transition</code> Service	47
9.5.3	Configuration	48
9.6	Output	48
9.6.1	The <code>output</code> Service	48
9.6.2	What Output Sends to <code>tutalkd</code>	49
9.6.3	Configuration	49
9.7	Student Model	49
9.7.1	Submodules	49
9.7.2	The <code>skip-optional</code> Query	50
9.7.3	The <code>sem-covered</code> Query	50
9.7.4	The <code>choose-goal-version</code> Query	51
9.7.5	Configuration	51
9.8	Dialogue Manager	52
9.8.1	Configuration	52
9.9	Dialogue History Manager	53
9.9.1	Public API	54
9.9.2	Configuration	54
9.10	Session Manager	54
9.10.1	Public API	55
9.10.2	Configuration	55



# 1 Installation

This section discusses the steps needed to get TuTalk up and running on a Unix-ish system.

## 1.1 Basic Requirements

Here are the things you need before you can even think of running TuTalk:

- Python 2.5 or later
- Cygwin, if you insist on running under Windows (not recommended)
- A C compiler, if you can't get binaries for the items below.
- The libxml2 C library<sup>1</sup>
- The libxml2-python Python module<sup>2</sup>
- The SQLite C library<sup>3 4 5</sup>

To install a Python package, you typically issue the command `python setup.py build` and then `sudo python setup.py install`.

## 1.2 Getting the Files for Local Users

If you are local to Pitt, this is how you get a local copy of the latest software and get the latest updates. The only prerequisite is an account on `unixs.cis.pitt.edu` and asking us to add you to the `tutalk` group. If you are not local to Pitt, you should instead download the tarball at <http://www.pitt.edu/~tutalk/TuTalk.tar.gz>.

In red are the commands you issue to get a local copy.

```
$ cvs -d :local:/afs/pitt.edu/home/t/u/tutalk/private/cvsroot tutalk  
(Enter your password.)  
cvs checkout: Updating .  
U AsyncoreThread.py  
U DialogueHistory.py  
U DialogueManager.py  
U Input.py  
<snip>
```

---

<sup>1</sup>Available from <http://xmlsoft.org/downloads.html> but you probably have it already if you run Linux or OS X 10.3+

<sup>2</sup>Available from <ftp://xmlsoft.org/libxml2/python/> or <http://xmlsoft.org/python.html> **OS X 10.4 users:** do not use the latest version! I had good results with version 2.6.15. OS X 10.5 will hopefully provide, or at least be compatible with, version 2.6.21.

<sup>3</sup>Available from <http://www.sqlite.org/> but you probably have it already if you run Linux or OS X 10.4+

<sup>4</sup>Statements by its enclosed docs to the contrary, it should be just a `./configure; make; sudo make install` kind of exercise. However, on Pam's computer (and subsequently on Why2) we had an interesting time trying to get the SQLite library `libsqlite3` recognized. If you have installed the library (typically in `/usr/local/lib`) and `ldconfig` still can't find it, make sure the file `/etc/ld.so.conf` contains the line `/usr/local/lib`. If you need to add that line, you may need also to execute the command `sudo /sbin/ldconfig` so the linker can update its database.

<sup>5</sup>If installing from source, and you get an error about TCL during the make stage, trying disabling it via `./configure --disable-tcl`.

To get subsequent updates, you can issue this command from within the sandbox:

```
$ cvs up -d
```

The `-d` flag instructs CVS to create any directories added to the repository since the last update. This is generally a good idea.

### 1.3 Run `setup.py`

Initial setup is done by running `setup.py`, located in the main TuTalk directory. It is responsible for compiling the C-language backends used by some of the Python code, and verifying that the local Python provides all the necessary built-in packages.

### 1.4 Optionally Setting up TuTalk as an HTTP Server

Optionally setting up `tutalk` as an http server is done by running `configureWebsite`, located in the main TuTalk directory. You do not need to do this in order to explore TuTalk.

```
$ sudo ./configureWebsite
```

This will set up the admin page for the TuTalk server and prompt you to generate accounts. Afterwards, you will need to start `tutalkd` by going to the main TuTalk directory and running `./tutalkd -d`. After running this you can access the TuTalk admin page using the `ADMIN_CGI` url that you specified and login using one of the accounts you just created.

Once you are logged in, you can upload, launch and quit scripts. Note that you need to refresh the page manually using the “update page” button near the bottom to see the effects of launching or quitting a script. Once you “quit” a script, you can view any transcripts collected and query the dialogue history database for that script using the “databases” link, and any logs in the “Log files” section. You can point the interface that you build for your application to this server and any script that is running on it (see the example interface in the subdirectory `client/Java`). Finally, you can also add additional accounts via the admin page using the “account maintenance” link.

The owner of the TuTalk directory can find all of these files in the Experimenters subdirectory. There is one subdirectory per account name. Each account has a “scenarios” subdirectory with the dialogue scripts that have been uploaded and a “logs” subdirectory with log files and the dialogue history database.

### 1.5 Running TuTalk from the Commandline

The TuTalk coordinator lives in the executable Python file `TuTalk.py`. It takes a single optional argument: the name of the XML scenario file to load. If none is provided, it uses `TestScenario.xml` which is a minimalistic script used for testing. This executable will also start up `tutalkd` as a separate process and it will continue until you explicitly end the `tutalkd` process. If all goes as hoped, your session should look something like this:

```
[moses@hal TuTalk]$ ./TuTalk.py
ALL MODULES READY (took 2.004693 seconds)
```

For more information on interacting with the Coordinator, including how to log in as a “fake” student from the command line, see Section 5.4.

## 2 Architecture

TuTalk is a collection of “modules” implemented in Python<sup>6</sup> which act as agents; their efforts are coordinated by a module called the Coordinator (the file `TuTalk.py`). Modules advertise a list of services they provide when they are started up. When running, they can put up requests for certain operations. The Coordinator dispatches these to appropriate modules (those that have advertised themselves as offering such a service). The Coordinator routes requests from the task list to modules, and adds requests from modules to the task list. All communication between modules is in the form of XML messages and go through the Coordinator.

*Example: the Dialogue Manager’s request for “normalized student input” goes to the Normalizer module, or whoever may advertise “normalized student input”. The Normalizer then, knowing that it needs student input in order to normalize it, creates a request for “student input”, which ends up on the Input Module’s agenda. The request stays there until it can return the appropriate data – until the student does something and the input can be returned for preprocessing and classification. Thus the student input travels back to the Normalizer, and when the Normalizer is done returns a reply to the Dialogue Manager, which will most like issue a request to the Natural Language Understanding (NLU) module for classification.*

The system runs asynchronously to support multiple students logged in simultaneously. The fact that different modules are cooperating processes decreases the likelihood that the system will become unresponsive even if one task requires significant machine resources.

Multiple copies of TuTalk are runnable without interfering with each other. Input and Output modules communicate with the outside world through an intermediary service called `tutalkd`.

One thing TuTalk can’t do is run two copies of the same scenario under the same user. The Coordinator uses a lockfile to make sure this can’t happen, although since race conditions are almost always possible with file I/O, you shouldn’t try to do this.

## 3 Dialogue Scenarios

We refer to an authored dialogue script as a “scenario” to highlight the fact that it is a self-contained specification for a “unit” or “problem” for our tutoring system. We emphasize that a scenario contains both the script (“recipes”, or sequences of conversation steps) and the natural language information to realize it.

There are two forms that such scenarios can take; both will be used in the examples in this section. The first is the XML format emitted by the CMU authoring tool. This is the format that TuTalk can read directly. The second format is read by a program called `sc` (“Scenario Compiler”) and converted to XML. We will use the term `sc` to refer both to the file format – one might call it a mini-programming-language – and to the program that processes it. Both XML and `sc` documents are required to be in UTF-8 Unicode, and are typically named with file suffixes `.xml` and `.sc` respectively. The first subsection discusses `sc`.

Dialogue scenarios have two types of required material, linguistic concept definitions (“concepts”) and recipe steps (“script”); they will be discussed in depth first. The two other types of material, module customization (“configuration”) and topic transitions/feedback (“transitions”), will be discussed afterwards since they are not as frequently used.

---

<sup>6</sup>They should not be confused with “Python modules” which are like libraries. TuTalk modules are separate programs; they run as separate processes.

The DTD (Document Type Definition) that defines the TuTalk XML scenario files fixes these information types in sections that come in a prescribed order. In `sc` the elements can be interleaved, but most authors will find it convenient to group them together in an order similar to the XML.

### 3.1 The `sc` Scenario Compiler

`sc`<sup>7</sup> is designed to be an easy way to throw together a small scenario file, and to make complex scenarios readable and maintainable. All features of the XML scenario format are exposed to authors. `sc` requires no special configuration other than a complete TuTalk installation, of which it is part.

`sc` is located in the `Compiler` subdirectory of the main TuTalk directory. We provide an emacs mode for syntax highlighting written by Scott Silliman; see the file `Compiler/sc-mode.el`. It contains installation instructions. *Note however that emacs does not handle Unicode correctly at this time. Linux editors such as `gedit`, and most applications on Mac OS X, are recommended for authoring foreign-language dialogues.* For `gedit`, and other Gnome/GTK applications we provide a syntax mode in `Compiler/sc.lang` which also contains installation instructions.

An `sc` file is divided into “chunks”, each of which contains either configuration, transition, concept, or script information. Chunks are separated by one or more blank lines. Anything after a pound sign (`#`) is a comment.

Here is an example of an extremely minimalistic scenario, which consists of two concept chunks and a single goal chunk that contains the entire (tiny) dialogue script, which is intended to be self-explanatory:

```
1 c greeting
2   "Hello."
3
4 c goodbye
5   "See you later."
6
7 g start
8   say greeting
9   if greeting say "Nice to meet you."
10  otherwise say "How rude!"
11  say goodbye
```

Perhaps the most interesting aspect of `sc` format, and the one that differentiates it the most from its equivalent XML, is that tutor and student strings can be used inline within the dialogue, as in the example with the “Nice to meet you.” and “How rude!” comments by the tutor. For some users this makes it easy to visualize the scripted dialogue without hunting around for the concept definitions.

As a script is refined, with multiple phrases per concept being added, the strings can be migrated off into concept chunks and fleshed out. This can be semi-automated with the use of the reciprocal `unsc` program, which produces `sc` from XML.

---

<sup>7</sup>Named in imitation of the Unix `cc` C compiler command.

## 3.2 Concepts

Concepts are unified dialogue contributions with similar semantic content. A set of alternative concepts for an expected reply to a dialogue contribution can (and will) be referred to as a language model (in the context of the given dialogue contribution).

Every concept has a concept label that is used to refer to it when specifying recipe steps. Note that these labels must be unique identifiers; that is the same label cannot be used for more than one concept definition. Doing so will result in errors in the runtime dialogue system.

In the example below, there are concepts defined with labels such as “no” and “skip-appetizer” which will be used when specifying dialogue recipes. Each concept is specified by listing the phrases that have the same concept meaning (e.g. the phrases “no”, “nope” and “I need a few minutes” are the phrases that define the concept “no”. All phrases for the concept meaning should be listed but care must be taken not to over-extend a concept since dialogue contributions are context sensitive. For example, the concept “no” as defined below would not be appropriate for a question such as “Do you like this restaurant?” since the phrase “I need a few minutes” is an inappropriate response. Whereas this particular concept definition is a perfectly acceptable response for “Are you ready to order?”. Just because “no” is an appropriate response for both questions does not mean that the two concepts can be merged.

A number of phrases for a concept can be combined by including only stems and non-function words (e.g. the phrase “I not appetizer” which would cover such phrases as “I do not want an appetizer”, “I would not like any appetizer”) but in most cases below full phrases are listed. When non-function words are omitted, the `phrase` attribute `minimal="yes"` must be used.<sup>8</sup>

When matching against student input, the student input is first “normalized” by a language “expert” plug-in in the Normalizer module. How this is done depends on the specific language. The supplied English normalizer (`normalizer/en-normalizer.py`) does case and spacing normalization, morphology and stopword removal, and spelling correction. The Chinese normalizer (`normalizer/zh-normalizer.py`) does word identification, introducing spaces to show word boundaries, which are needed by the NLU module’s Levenshtein matcher.

Again, care must be taken when specifying minimal phrases to consider the other phrases included in the concept and the contexts in which the concept will be used. For example, in the Why2-Atlas system, one expected response concept for “what do you know about the magnitude of its displacement from the center of the pool when it lands? Can it be zero? Or must it be non-zero?” included the concept “no” and too many phrases from too many other concepts had been aggregated with it so that both “no” and “zero” were specified as acceptable minimal phrases. A student replied “zero” and this was misrecognized as the concept “no”. The aggregation of “no” with “zero” should have instead used the minimal phrase “no change” to suit a context such as “What is the change in velocity?” but because the goal was to create a minimal phrase for “no difference” “no change” “no changes” etc, only “no” was retained. Thus it was not intended to be the same “no” as in a “yes/no” response.

By default a concept does not express a transition between topics or turns. If transitions are configured to be enabled, the system will automatically generation dialogue transitions (see later section on transitions). However, an author can override the automatically generated transitions by defining their own transition concepts. These concepts are indicated by setting the “transition”

---

<sup>8</sup>Care should be taken, especially when mixed initiative is enabled, to provide at least one fully-specified phrase for the concepts the system can emit.

attribute to “yes” (default is “no”) when defining a concept. Whenever this transition concept appears in the script, it will suppress contiguous automatic transition generation. See the concept “fine-then”, defined below, for an example.

The scenario configuration needs to specify the default language for phrase strings. Any phrase not in the default language must be given a `lang="xy"` attribute, where `xy` is an ISO 639.1 language code<sup>9</sup> (case insensitive). For the concept `skip-appetizer` below, the phrase *Je n'ai pas le temps pour un apéritif* is identified as French.

Specific phrases may require different or additional processing beyond what the NLU module provides. For example, the phrase “( \_ |”<sup>10</sup> below in the definition of the concept `skip-appetizer` may be better handled by a special emoticon-recognizing submodule as indicated by `flag="emoticon"`. Similarly, equations or long explanations may require special processing as was done in the Why2-Atlas system.<sup>11</sup>

The TuTalk deliverable includes one such specialized processor, for communicating with an off-site TagHelper server.<sup>12</sup> Its implementation file is in the `nlu/taghelper.py` directory and it may be used as a starting point for submodule development. Phrases flagged for TagHelper are in fact left empty, as the data to be matched against is in fact in the form of a classifier running as a separate program (in our case on another machine). The classifier/class data is in this particular case embedded in the concept name. The built-in matcher doesn't even try to use any of the flagged phrases; these are strictly for the TagHelper submodule.

```
1 <concept label="soc.0:socagg_yichiaw">
2   <phrase flag="taghelper"></phrase>
3 </concept>
4 <concept label="soc.201:socagg_yichiaw">
5   <phrase flag="taghelper"></phrase>
6 </concept>
7
8
9 c soc.201:socagg_yichiaw
10  "" flag taghelper
11
12 c soc.0:socagg_yichiaw
13  "" flag taghelper
```

Note that any other special processing associated with settings of the attributes `lang` and `flag` is not provided as part of the TuTalk deliverable. Applications or projects that need special processing support must provide these submodules by either finding existing software from elsewhere and writing the necessary “glue” (submodule) to plug into the Tutalk NLU or Normalizer architecture<sup>13</sup>

---

<sup>9</sup><http://www.w3.org/WAI/ER/IG/ert/iso639.htm>

<sup>10</sup>If you squint carefully you will see this represents being “moonied” by the emoticon’s user. We saw this one in an actual experiment with an earlier system (Why2-Atlas) and it somehow managed to crash our server, which seems to have been deeply offended by the gesture.

<sup>11</sup>In Why2-Atlas long explanation phrases are actually pointers to the knowledge representations for the expected response (e.g. “p30a p31b”).

<sup>12</sup>See <http://www.cs.cmu.edu/~cprose/TagHelper.html>

<sup>13</sup>Some adaptation of an existing software is likely to be needed in order to plug it in to the TuTalk architecture and it is the responsibility of the application or project itself to fund the necessary adaptations.

or by allocating resources to develop the necessary submodule.

```
1 <concepts>
2   <concept label="no">
3     <phrase>no</phrase>
4     <phrase>nope</phrase>
5   </concept>
6   <concept label="enthusiasm-about-appetizers">
7     <phrase>Hey, I'll bet they have some really good appetizers.</phrase>
8   </concept>
9   <concept label="ask-appetizer">
10    <phrase>Let's look at the appetizers.</phrase>
11    <phrase difficulty="ynq">So, should we choose an appetizer first?</phrase>
12    <phrase difficulty="ynq">Do you want to get an appetizer?</phrase>
13    <phrase difficulty="whq">What do you want for an appetizer?</phrase>
14  </concept>
15  <concept label="skip-appetizer">
16    <phrase minimal="yes">I not appetizer</phrase>
17    <phrase>I never eat appetizers</phrase>
18    <phrase>I can't afford any</phrase>
19    <phrase>I'm not very hungry</phrase>
20    <phrase>I don't have time for an appetizer</phrase>
21    <phrase flag="emoticon">(_(|</phrase>
22    <phrase lang="fr">Je n'ai pas le temps pour un apéritif</phrase>
23  </concept>
24  <concept transition="yes" label="fine-then">
25    <phrase>Well, uh... OK</phrase>
26    <phrase>Fine. Be that way!</phrase>
27    <phrase>Okay, if that's what you want.</phrase>
28  </concept>
29  <concept label="sounds-good">
30    <phrase>Sure</phrase>
31    <phrase>Sounds good</phrase>
32    <phrase>OK</phrase>
33  </concept>
34 </concepts>
35
36
37 c no
38   "no"
39   "nope"
40
41 c enthusiasm-about-appetizers
42   "Hey, I'll bet they have some really good appetizers."
43
```

```

44 c ask-appetizer
45 "Let's look at the appetizers."
46 "So, should we choose an appetizer first?" difficulty ynq
47 "Do you want to get an appetizer?" difficulty ynq
48 "What do you want for an appetizer?" difficulty whq
49
50 c skip-appetizer
51 "I not appetizer" minimal
52 "I never eat appetizers"
53 "I can't afford any"
54 "I'm not very hungry"
55 "I don't have time for an appetizer"
56 "(_(|" flag emoticon
57 "Je n'ai pas le temps pour un apéritif" lang fr
58
59 c fine-then trans
60 "Well, uh... OK"
61 "Fine. Be that way!"
62 "Okay, if that's what you want."
63
64 c sounds-good
65 "Sure"
66 "Sounds good"
67 "OK"

```

### 3.2.1 Concepts in sc

A concept chunk starts with a line “c [concept-name]”, where the concept name is optional. If omitted `sc` will create a concept name by concatenating together some words of one of its phrases if possible, and failing that, it will use a generic numbered name, such as “concept-101”.

The subsequent lines are quoted phrases followed by attributes:

- `difficulty`, `lang`, and `flag` require a following argument.
- `minimal` is just a binary flag and takes no parameter.

#### Example:

```

1 c lakes
2 "superior"
3 "huron"
4 "erie"
5 "michigan"
6 "ontario"
7
8 c correct
9 "Good."

```

Here is a silly example that illustrates all the features.<sup>14</sup>

```
1 c a-russian-concept transition
2 "Смерть шпионам" diff impossible lang ru
3 flag some-flag
```

### 3.3 Recipes

Recipe steps are used by the APE-based dialogue manager module. Every recipe has a goal name which need not be unique and will be referenced when invoking any subdialogue in response to the dialogue partner replies (as will be seen later in the example below).

The initial goal is named **start**. There must be at least one **start** goal in a scenario file, otherwise TuTalk will complain and quit.

**Difficulty labels.** An optional difficulty label can be supplied to provide additional context that will support choosing between multiple versions of recipes with the same goal name. If no difficulty label is supplied the default behavior is to choose the least recently used recipe for the goal name. If difficulty labels are supplied then a supporting student model policy must be configured for the dialogue system. The difficulty label and decision policy are of the authors own choosing. For example, the difficulty labels can be numeric and represent difficulty levels so that the student model examines the dialogue history and the student's past performance and chooses either an easier or harder version of the recipe than was previously chosen. Alternatively, a symbolic labelling scheme and a policy for choosing between the symbolic labels can be used (as will be seen with utterance level choices).

```
<goal name="start"
  <step>
    <subgoal>ask-appetizer</subgoal>
  </step>
</goal>

<goal name="ask-appetizer" difficulty="short-version">
  <step optional="yes">
    <initiation sem="enthuse-about-appetizers">enthuse-about-appetizers</initiation>
  </step>
  <step>
    <initiation sem="elicit-appetizer-order">ask-appetizer</initiation>
    <response push="abort soup">skip-appetizer</response>
    <response say="fine-then" push="abort soup">no</response>
    <response push="lose-temper">unanticipated-response</response>
  </step>
  <step>
    <initiation>order-appetizer</initiation>
  </step>
```

---

<sup>14</sup>The phrase means “death to spies”; it is hard to imagine how such a thing could be used as a transition in a serious dialogue unless the tutor wanted to say this as negative feedback in case the tutee admitted to moonlighting in clandestine ops.

</goal>

**Recipe steps.** Once the recipe is identified and any optional difficulty labels specified, then each step of the recipe is specified. A step is a pairing of an initiation and expected responses, if any. Steps do not have labels in order to discourage authors from jumping to the middle of a recipe. If part of a recipe is to be referenced and reused in another recipe then that part should be defined as a separate recipe and the replacement step is then simply a pointer to the sub-recipe (as illustrated below under “Embedding a recipe in another”).

**Initiations.** An initiation indicates what concept to express (in the example above, enthuse-about-appetizers, ask-appetizers and order-appetizer) to initiate the step. An initiation can be assigned an optional semantic label (“sem”) attribute where the basic intent is for steps across recipes that have similar meaning to carry the same semantic label. The semantic label is of the author’s own choosing but an appropriate student model policy must be configured.

When a step is reached where a choice is needed in either the initiation or response, then the dialogue manager module will identify the available choices and will request a decision. The student model module advertises the ability to fulfill such decision requests. Currently, there are two types of choices with respect to step parts 1) a step is specified as optional 2) multiple ways of realizing an initiation or response or subgoal are specified. In both cases, a semantic label must have been specified for the relevant part of the step to support the student model’s decision making process.

**Optional steps.** If a step is optional (in the example above, optional=“yes”) then an example policy for the student model is to skip the step if the semantic label on the initiation (in the example above, sem=“enthuse-about-appetizers”) appears in the recent dialogue history or to skip the step if the semantic label appears less recently in the dialogue history but was answered appropriately or is inappropriate to repeat (e.g. was said by the dialogue partner). So if the recipe “ask-appetizer” has been invoked recently for the language-learning student and is to be invoked again then the semantic label “enthuse-about-appetizers” will be in the dialogue history and the optional step with label “enthuse-about-appetizers” will get skipped on the current invocation of the recipe.

Alternatively, optional=“said-once” will cause the step to be skipped if the sem label for the initiation ever appears in the dialogue history for the student. This allows some small adjustments in the decision policy for when to skip.

**Responses.** Note that for the initiation in the first step of the above example recipe for “ask-appetizer” that no response is anticipated but for the initiation in the second step, some anticipated responses are specified. As with initiations the previously specified concept labels are put to use. The main difference between initiation and response specifications is that more than one response concept can be specified. The “push” attribute value indicates the recipe for a subdialogue that is to be invoked in response if the natural language understanding module determines that the language-learning student’s response best fits the concept listed. The push entry may contain multiple recipes, delimited by whitespace.

For example, if the “no” concept is the best match for the student’s response to the system then the dialogue manager will invoke the recipe “soup” (not shown here) and once that recipe is completed the dialogue manager will return to the next step of the recipe that originated the just completed recipe (in this case it will return to the “ask-appetizer” recipe but since there are no unrealized steps remaining to complete, the recipe will be complete) unless abort was indicated.

In the example above, the response to “no” also specifies a system comment in addition to

the “abort” pseudo-recipe. The “say” attribute refers to a concept label from which a string is generated. The purpose of this is to make it easier for authors to customize short tutor comments without the need to author a larger block of dialogue. These system comments are always presented first and then any push actions that are indicated are taken. In this case the system emits “Well, uh...OK” in response to the curt “no” response, and then terminates the “ask-appetizer” recipe due to the presence of the “abort” pseudo-recipe.

A standard concept label of “unanticipated-response” should always be included and an appropriate response recipe specified. As many concepts as are needed can be listed as anticipated responses. The “unanticipated-response” choice is followed only if none of the expected response concepts matched. If student initiative is allowed according to the configuration, then there is the additional restriction that no show of initiative (see definition below in section on student initiative) is detected before the “unanticipated-response” choice is selected.

An additional built-in recipe (like “abort”) is “logout”. This causes the tutor to inform the student that he or she is logged out, and then end the session. This is useful when a scenario is large/long enough that it is appropriate to allow a student to end a session and resume at a later date or time.

**Effect of difficulty labels on initiations and responses.** If alternative ways of realizing a concept are defined for an initiation or response then an example policy for choosing between them is to check which category of utterance was last selected and if answered appropriately to choose the next hardest version of the utterance. If it was answered inappropriately then an example policy is to choose the next easiest version of the utterance.

For example, four alternatives for soliciting an appetizer order are available for the script example above (concept ask-appetizer in recipe ask-appetizer). The type of each alternative utterance is specified with the attribute “difficulty” in its concept entry (above). The default setting of values are ynq (for yes-no question), inform (for an inform speech act – i.e. a simple assertion or statement), whq (for where, what, when questions), whyq (for why questions). The set of values can be selected by the author but the student model policy must be configured for these values and how to use them during decision making.

In the example script above, we will assume that a yes-no question (ynq) is easier for a language-learning student to answer, while the what question (whq) and the inform are progressively harder in that the first requires more language generation and the second even more. So an example policy is to first try the yes-no question (ynq) for a student (e.g. “So, should we choose an appetizer first?”) and then if another practice is done or the response to the question was “no” and the question is revisited later, to next try the harder question if the student did well with the first (e.g. “What do you want for an appetizer?”). Doing well means either that the student responded appropriately or answered correctly.

**Truth values.** Initiations and responses can also include a `truth-val` attribute that is useful for factual material as opposed to simply chatting or expressing opinions (in the latter cases it is generally assumed that the speaker does not say anything he believes to be false and we will assume that the system has no way to assess the validity of non-factual material). A value of “yes” indicates that the concept is true in the current context, “no” indicates that it is not, “partial” indicates that it is vague and at least potentially true, and “unknown” is for concepts that do not express factual material. The `truth-val` attribute is used in generating acknowledgements of what the student just said and for selecting how to respond to an initiation (if the goal is for the system to always be truthful).

We will move to different example dialogue scripts to illustrate the remaining script features which include embedding a recipe in another, requiring multiple responses for a step, and retrying a recipe (in effect implementing a loop). Finally, we will end with a discussion of the system's behavior with respect to student initiative.

**Embedding a recipe in another.** A recipe can embed another recipe by referring to the goal name of that recipe in a step. For example, in the script below, the recipe "order-meal" embeds four other recipes as steps.

```
<goal name="order-meal" difficulty="1" sem="restaurant-meal">
  <step optional="said-once">
    <subgoal>appetizer</subgoal>
  </step>
  <step optional="said-once">
    <subgoal>salad-and-soup</subgoal>
  </step>
  <step optional="said-once">
    <subgoal>entree</subgoal>
  </step>
  <step optional="said-once">
    <subgoal>dessert-and-coffee</subgoal>
  </step>
  <step>
    <initiation>bon-appetit</initiation>
  </step>
</goal>
```

The effect is that a push is done to the first embedded recipe "appetizer" and when that recipe completes, control returns to "order-meal" and a push is done to "salad-and-soup". A sem label can be associated with a recipe (e.g. restaurant-meal on the recipe "order-meal" above) as well as the parts of a step. Assuming that the recipes "appetizer", "salad-and-soup" and "dessert-and-coffee" each have sem labels defined for each recipe then these optional steps (see optional="said-once") will be skipped if the sem label for a recipe appears in the discourse history (see below for an example recipe for "appetizer" and for "salad-and-soup").

This example assumes that student initiative is enabled (see later subsection on Student Initiative). Thus a student could interrupt a topic such as the one covered in the subgoal "entree" to discuss instead "dessert-and-coffee". When the dessert-and-coffee recipe is finished then the system will try to return to the partially completed "entree" recipe and if that recipe completes then it would attempt the "dessert-and-coffee" recipe except that the optionality of this step is set to "said-once" and the dialogue history indicates that this recipe had previously been initiated. Thus it will skip it and go on to the next step.

*Pam notes: What if the student interrupts the "dessert-and-coffee" recipe and returns to the "entree" recipe?*

**Requiring multiple response parts in a step.** An initiation can also result in an expectation of multiple concepts in response to the one initiation as in the example below.

```
<goal name="salad-and-soup" difficulty="1" sem="want-salad-and-soup">
```

```

<step>
  <initiation necessary-and-sufficient-answer="soup-or-not salad-or-not">
    do-you-want-soup-and-salad
  </initiation>
  <response push="ack-soup"
    say-nomatch="you-not-answer-all-of-my-question"
    push-nomatch="what-about-soup">
    soup-or-not </response>
  <response push="ack-salad" push-nomatch="what-about-salad">
    salad-or-not </response>
  <response say="you-not-listening-to-me">
    unanticipated-response </response>
</step>
</goal>

```

Assuming the concept “do-you-want-soup-and-salad” includes a phrase such as “Do you want soup and salad?” and excludes a phrase such as “Do you want soup or salad?”, then it sets up a discourse obligation to answer about both soup and salad. Although it is possible to set up specialized concepts for a single concept that combine answers about both soup and salad this is dispreferred because if the student answers only about soup or only about salad then the student will not be given “partial” credit for an appropriate partial response.

To signal that a multi-part response is anticipated the **necessary-and-sufficient-answer** attribute must be included in the step specification. All the concepts that represent a complete answer should be listed in the answer specification. Note that even if an answer tag is not included, it is possible to recognize multiple concept matches and perform whatever actions are indicated for each match.<sup>15</sup> The difference is that one response match is necessary and sufficient whereas with multi-part responses one response match is not sufficient. The response attributes of “push-nomatch” and “say-nomatch” are similar to “push” and “say” but only apply to multi-part answers and specify the actions to take when a **required** answer part is missing. If the answer concept is not listed in the “answer” tag then it is not required and the nomatch attributes will be ignored.

The unanticipated-response concept matches only if none of the required answer concepts matched. In the case of no answer specification then unanticipated response is matched only if none of the responses matched. If the system is configured to allow student initiative then “unanticipated-response” is followed only when all required matches fail and no show of initiative is detected.

**Retrying (looping) of a recipe.** An entire recipe can be made to loop until a specified condition is met. We do not provide a direct means of looping on just part of a recipe. If the latter is desired then it can be accomplished by specifying the part that is to loop as a separate recipe and specifying just this extracted recipe to loop, and then including the new recipe as a single step in the original recipe.

A recipe that is to loop is specified by including the goal attribute “retry-until-cover” which

---

<sup>15</sup>Note that multi-part response and cases of looping on a recipe (see retry-until-covered) both are cases where we strongly expect multiple matches. In the other cases, it is less strongly expected but not disallowed. Note that only the best match for a span of a response is used so multiple matches apply only to different spans of a response.

encodes the termination condition. In `sc` it's called "loop". If no termination condition is specified then the goal will not loop. The termination condition is limited to the sem labels that must appear in the dialogue history. All sem labels included in the condition must appear in the dialogue history before the loop will terminate. **Warning:** there is **no** checking or validation of the sem labels that appear in a condition. This means that if a sem label in a condition never appears anywhere in the script then an **infinite loop** of the goal may occur.

The example goal below (some related concept definitions follow after) illustrates the specification of a recipe that will loop until the "finished-appetizer" sem label appears in the dialogue history or the concept associated with an "abort" action is matched.

```
<goal name="appetizer" difficulty="1" sem="appetizer"
  retry-until-cover="finished-appetizer">
  <step optional="said-once">
    <initiation sem="open-appetizer-topic">
      suggest-look-at-appetizers</initiation>
    </step>
    <step>
      <initiation>what-appetizer-looks-good</initiation>
      <response push="discuss-calamari">
        interested-in-calamari</response>
      <response push="discuss-appetizer2">
        interested-in-appetizer2</response>
      <response push="discuss-appetizer3">
        interested-in-appetizer3</response>
      <response sem="finished-appetizer">no-more-appetizers</response>
      <response say="ack-skip-appetizer"
        sem="finished-appetizer">not-want-appetizer</response>
      <response push="abort">unanticipated-response</response>
    </step>
  </goal>
```

```
g appetizer difficulty 1 sem appetizer loop finished-appetizer
say suggest-look-at-appetizers once sem open-appetizer-topic
say what-appetizer-looks-good
if interested-in-calamari do discuss-calamari
if interested-in-appetizer2 do discuss-appetizer2
if interested-in-appetizer3 do discuss-appetizer3
if no-more-appetizers sem finished-appetizer
if not-want-appetizer say ack-skip-appetizer sem finished-appetizer
else do abort
```

Note that for the concept definitions below we used numeric difficult labels that encode the difficulty-levels of phrases. But in this case we used the difficult labels to implement initial vs subsequent use phrases. The first time the initiation for the second step is executed `difficulty='1'`

is selected and on subsequent executions of this step `difficulty='0'` is selected (since 0 is the “sink” for the selection policy).

```
<concepts>
  <concept label="suggest-look-at-appetizers">
    <phrase>Let's look at the appetizers.</phrase>
    <phrase>Let's order some appetizers.</phrase>
  </concept>
  <concept label="what-appetizer-looks-good">
    <phrase difficulty="1">What appetizer looks good to you?</phrase>
    <phrase difficulty="1">Which appetizers look good to you?</phrase>
    <phrase difficulty="0">What other appetizer looks good to you?</phrase>
    <phrase difficulty="0">Which other appetizers look good to you?</phrase>
  </concept>
  <concept label="interested-in-calamari">
    <phrase>The calamari looks interesting.</phrase>
    <phrase>I like calamari.</phrase>
  </concept>
</concepts>
```

```
c suggest-look-at-appetizers
  "Let's look at the appetizers."
  "Let's order some appetizers."

c what-appetizer-looks-good
  "What appetizer looks good to you?" difficulty 1
  "Which appetizers look good to you?" difficulty 1
  "What other appetizer looks good to you?" difficulty 0
  "Which other appetizers look good to you?" difficulty 0

c interested-in-calamari
  "The calamari looks interesting."
  "I like calamari."
```

Alternatively, although cumbersome, two recipes could have been written to achieve the distinction of initial vs. subsequent asking of a question as below.

```
<goal name="appetizer" difficulty="1" sem="appetizer">
  <step>
    <initiation sem="open-appetizer-topic">
      suggest-look-at-appetizers</initiation>
    </step>
  <step>
    <initiation>what-appetizer-looks-good-initial</initiation>
    <response push="discuss-calamari">
```

```

        interested-in-calamari</response>
    <response push="discuss-appetizer2">
        interested-in-appetizer2</response>
    <response push="discuss-appetizer3">
        interested-in-appetizer3</response>
    <response sem="finished-appetizer">no-more-appetizers</response>
    <response say="ack-skip-appetizer" push="abort"
        sem="finished-appetizer">not-want-appetizer</response>
    <response push="abort">unanticipated-response</response>
</step>
<step>
    <subgoal>appetizer-subsequent</subgoal>
</step>
</goal>

```

```

<goal name="appetizer-subsequent" difficulty="1" sem="appetizer"
    retry-until-cover="finished-appetizer">
    <step>
        <initiation>what-appetizer-looks-good-subsequent</initiation>
        <response push="discuss-calamari">
            interested-in-calamari</response>
        <response push="discuss-appetizer2">
            interested-in-appetizer2</response>
        <response push="discuss-appetizer3">
            interested-in-appetizer3</response>
        <response sem="finished-appetizer">no-more-appetizers</response>
        <response say="ack-skip-appetizer"
            sem="finished-appetizer">not-want-appetizer</response>
        <response push="abort">unanticipated-response</response>
    </step>
</goal>

```

```

g appetizer difficulty 1 sem appetizer
say suggest-look-at-appetizers sem open-appetizer-topic
say what-appetizer-looks-good-initial
if interested-in-calamari do discuss-calamari
if interested-in-appetizer2 do discuss-appetizer2
if interested-in-appetizer3 do discuss-appetizer3
if no-more-appetizers sem finished-appetizer
if not-want-appetizer say ack-skip-appetizer do abort sem finished-appetizer
else do abort
do appetizer-subsequent

```

```

g appetizer-subsequent difficulty 1 sem appetizer loop finished-appetizer

```

```

say what-appetizer-looks-good-subsequent
if interested-in-calamari do discuss-calamari
if interested-in-appetizer2 do discuss-appetizer2
if interested-in-appetizer3 do discuss-appetizer3
if no-more-appetizers sem finished-appetizer
if not-want-appetizer say ack-skip-appetizer sem finished-appetizer
else do abort

```

**Student initiative.** Providing a mechanism for recognizing and responding to student initiative is out of the scope of the project that was funded. However, we agreed to provide a very crude means of responding to student initiative in order to explore the potential of chat dialogues for language learning. We define student initiative as any part of a student’s response that matches a concept outside of the current step that was not previously recently matched according to the dialogue history (i.e. may just be repeating what said earlier for coherency instead of meaning it to be an initiative). Currently the system will either always accept or reject an initiative depending on how the system is configured. No overriding of the set policy of either always accept or always reject is currently possible. Deciding to override the policy is out of scope for the currently funded project. First we will describe what happens when the policy is always accept or always reject. Then we will discuss potential problems that are anticipated with not allowing overrides of the policy and possible solutions to consider should the problem become a high enough priority that additional funding is allocated to address it.

If the default policy in the configuration is to accept any initiative, the current system will simply push to that initiative and when the initiative is completed it will resume whatever was interrupted. If some concepts within the current step are matched those will be pursued first and then the initiative will be taken up. The transition from completed initiative back to what was interrupted has the potential to lead to incoherencies in the dialogue.

**Potential problems and possible future enhancements related to student initiative.** The problem with an accept all or none policy is that it may sometimes be more appropriate to override the default policy. In the case of always accepting student initiative, it may sometimes be more appropriate to ignore or postpone the initiative depending on the relative importance of the current step and the initiative. In the case of always ignoring initiative it may sometimes be more appropriate to accept or postpone the initiative. Again it would depend on the relative importance of the current step and the initiative.

In addition to knowing when to override the default, it is tricky to properly implement postponing or accepting initiative. To properly postpone, perhaps the system should offer an acknowledgment of the initiative that communicates it has been understood and will be entertained once the current topic is completed such as “first answer my question and then we’ll talk about <topic>”. Then it should put the postponed initiative as a goal on the dialogue manager’s stack and then retry the current step. Deciding where to put the postponed initiative on the stack is not straightforward either.

Simply interrupting the current goal to accept an initiative is also a potential problem. The proper acceptance of an initiative means deciding whether to push a retry of the current goal or some higher level goal on the dialogue manager stack before pursuing the initiative or instead completely pruning the current goal or some higher level goal from the stack.

A simple enhancement such as including an initiative flag for the step with values of “ignore”,

“postpone”, “accept” to override the global policy is insufficient as it only reflects the importance of the current step. To consider the relative importance of the initiative we would need to allow authors to specify a separate relative importance for sem labels or add an importance attribute to concept definitions as in the example below. If the global policy is to ignore initiative then if the relative importance of the two sem tags or concepts for the step initiation and the initiative match indicates that the initiative is more important than the current step then the global policy could be overridden and the initiative taken up by the system and the current goal either aborted or retried at a later time. The inverse would apply if the policy is to accept all initiatives. If relative importance cannot be decided then the global policy would continue to be followed.

```
<concept label="annoying-pickup-line" importance="1">
  <phrase>So, do you come here often?</phrase>
</concept>
<concept label="emergency" importance="10">
  <phrase>I have to go to the bathroom.</phrase>
  <phrase>My grandmother just stole a schoolbus full of penguins.
    Gotta go!</phrase>
</concept>
```

We stress again that *no mechanisms are available for overriding the initiative policy*; the above just presents some ideas for what would be involved in providing such a capability. If users’ find a strong need for more refined handling of initiative then we would need extra funding to provide such a capability. Not only would we have to provide a way for authors to specify relative importance, we would also have to design and implement mechanisms for postponing, accepting or ignoring given the relative importance.

### 3.3.1 Recipes in sc

A recipe is identified by a <goal> element in XML, so *sc* identifies these using the abbreviation ‘g’. Any *sc* file needs at least one goal chunk, for the *start* goal required for all scenarios.

A goal chunk’s first line looks like this: “g goal-name [attributes...]” The optional attributes are *difficulty*, *sem*, and *loop*<sup>16</sup>. All of them take an argument; *loop* will need a quoted argument if it is to contain multiple *sem* tags.

If you want to put a comment (one that is preserved in the XML) in the goal, use lines starting with the “%” or *comm[ent]* keyword after the initial line.

The recipe proper begins after the initial line and comments, if any. Each line is an initiation, a response, or a subgoal. *sc* is smart enough to determine where the step boundaries are. An initiation or subgoal signals the beginning of a step, so authors need not to specify this.

- *say* or *initiation*, followed by a quoted string or a concept name, followed by optional attributes.
- *if* or *response*, followed by a quoted string or a concept name, followed by optional attributes.
- *else* or *otherwise* or *unant[icipated]*, indicating XML’s *unanticipated-response*, followed by optional attributes.

---

<sup>16</sup>an abbreviation of the XML attribute *retry-until-cover*

- do or subgoal, followed by a goal name.

do and say can also be used as attributes, for XML's push and say attributes.

Most of the step part attributes are named the same as in the XML. But a few behave differently:

- answer stands for necessary-and-sufficient-answer. It takes multiple phrase arguments as long as they are all quoted, and it stops adding them when it gets an unquoted directive/keyword or the end of line. See below for an example.
- There is no truth-val; use the keywords {true,false,partial}.
- XML push is do as mentioned above.
- XML push-nomatch is do-nomatch

To specify a step as optional, put a keyword in the say (initiation) attributes:

- once corresponds to <step optional="said-once">
- opt[ional] corresponds to <step optional="yes">

#### Example:

```

1  g start
2    do lakes
3    do capitals
4    say "End of dialogue."
5
6  g capitals
7    say "What is the capital of Illinois?"
8    if "springfield" say correct
9    if "chicago" say "Chicago is a big city, but not the capital." do hint1
10   else do hint1
11
12  g hint1
13    say "Here's a hint: It has two syllables."
14    if "springfield" say correct
15    else do hint2
16
17  g hint2
18    say "Here's another hint: The first syllable is the name of a season."
19    if correct say correct do hint1
20    else say "The capital of Illinois is Springfield."
21
22  g lakes
23    say "Can you name one of the Great Lakes?"
24    if lakes say correct
25    else say "One is Lake Superior."
26    say "Can you name another?"

```

```

27   if lakes say correct
28   else say "Another is Lake Michigan."
29   say "Can you name one more?"
30   if lakes say correct
31   else say "The five Great Lakes are Superior, Michigan, Huron,
32         Erie and Ontario."

```

After `sc` finishes the XML conversion, it will ask you for rankings on recipe and phrase difficulty levels, if you used more than one of either. Then the XML output is run through the `xmllint` program for validation.

### 3.4 Transitions

Transitions are the transitional phrases to use when moving between turns or topics. Turn transitions comprise acknowledging or giving feedback that is not content-specific. Transition phrases that have the same attribute values associated with them can be grouped together under the tag “transition”.

A default set of transition phrases are provided for the TuTalk system and users are encouraged to adjust them to meet the needs of their particular application and genre of dialogue. A fragment of the default transitions is shown below.

```

1 <transitions>
2   <transition ack-type="agree"
3   floor-status="neutral">
4     <tphrase> yes </tphrase>
5     <tphrase> okay </tphrase>
6   </transition>
7   <transition ack-type="agree" topic-status="continue"
8   floor-status="grab" scope="nonimmediate">
9     <tphrase> Yes, I agree. </tphrase>
10    <tphrase> Very good! </tphrase>
11    <tphrase> Excellent! </tphrase>
12  </transition>
13  <transition ack-type="agree" ack-polarity="neg">
14    <tphrase> I disagree with you. </tphrase>
15    <tphrase> That doesn't sound right to me.</tphrase>
16  </transition>
17  <transition ack-type="understand" floor-status="neutral">
18    <tphrase> I understand what you are saying. </tphrase>
19    <tphrase> I think I understand what you mean. </tphrase>
20  </transition>
21  <transition ack-type="hear"
22    floor-status="concede" scope="immediate">
23    <tphrase> uh-huh </tphrase>
24    <tphrase> hmmm-huh </tphrase>
25    <tphrase> yeah </tphrase>

```

26 <tphrase> I'm listening. </tphrase>  
27 <tphrase> Go on.</tphrase>  
28 </transition>  
29 <transition topic-status="refresh">  
30 <tphrase> So, back to the original question. </tphrase>  
31 <tphrase> Let's try the original question again. </tphrase>  
32 <tphrase> Once again on the original question. </tphrase>  
33 </transition>  
34 <transition topic-status="interrupt">  
35 <tphrase>  
36 Let's put this aside for a minute and come back to it later.  
37 </tphrase>  
38 </transition>  
39 </transitions>  
40  
41  
42 t agree neutral  
43 "yes"  
44 "okay"  
45  
46 t agree nonimmediate  
47 "Yes, I agree."  
48 "Very good!"  
49 "Excellent!"  
50  
51 t agree neg  
52 "I disagree with you."  
53 "That doesn't sound right to me."  
54  
55 t understand neutral  
56 "I understand what you are saying."  
57 "I think I understand what you mean."  
58  
59 t hear concede immediate  
60 "uh-huh"  
61 "hmmm-huh"  
62 "yeah"  
63 "I'm listening."  
64 "Go on."  
65  
66 t refresh  
67 "So, back to the original question."  
68 "Let's try the original question again."  
69 "Once again on the original question."  
70

71 t interrupt

72 "Let's put this aside for a minute and come back to it later."

The "transition" attributes are "ack-type", "ack-polarity", "topic-status", "floor-status" and "scope".

The allowed acknowledgment types ("ack-type") are "agree", "understand" and "hear". The default value is "none" which is to be used when the transition phrases are topic transitions instead of acknowledgments. "Agree" is a phrase that is issued when the system is able to determine whether it agrees or disagrees with the dialogue partner.

Acknowledgment phrases have the additional attribute "ack-polarity" which is "pos" (default) for phrases that show e.g. agreement and "neg" for e.g. disagreement. The polarity only has meaning when there is an "ack-type" other than "none".

The "topic-status" attribute reflects which phrases are most appropriate relative to the current topic status. The default for most turn transitions is to "continue" the current topic. The other "topic-status" values indicate phrases that can be used to help communicate changes in topic. "New" is for a phrase that will introduce a new topic, e.g. "So let's talk about something new.", "interrupt" for one that will communicate that a topic is to be put on hold before moving to something new, "resume" for going back to a topic that was put on hold, "repeat" for knowingly repeating a topic or turn e.g. "Let me ask you this again.", "refresh" for repeating a topic to bring it into focus e.g. "Let's go back to something from before just to refresh your memory.", and "finish" for communicating that a topic is being closed e.g. "So we're finished with that now."

The "floor-status" attribute indicates whether a transition is more likely to be interpreted as interrupting the current speaker (i.e. grabbing the floor "grab" which is the default) or not ("concede") or whether it is indeterminate ("neutral"). Acknowledgments of hearing, such as "uh-huh", when uttered with the right prosody do not interrupt because they do not signal a grab for the floor (called backchannels in some dialogue literature). But acknowledgments that signal understanding or agreement are more contentful and tend to interrupt and thus often have the effect of "grabbing" the floor from the current speaker. Note that acknowledgments of hearing are only appropriate with spoken dialogue that allows barge-in so that these acknowledgments are made at a unit that may be smaller than a phrase. Acknowledgments of understanding are also better with spoken dialogue with barge-in allowed since these acknowledgments are generally at the sentence level. Acknowledgments of agreement are more likely to occur at the turn level and thus can be appropriate for both spoken and typed dialogue.

With acknowledgments, the "topic-status" is usually to "continue" the current topic (the default) and the "floor-status" is to "grab" the floor (the default).

"Scope" indicates whether the transition refers to just the most "immediate" dialogue or to a larger chunk of dialogue ("nonimmediate"). "Any" is the default for the "scope" attribute and indicates that the phrase is appropriate in either context.

To summarize the default settings are "ack-type" = none, "ack-polarity" = pos, "topic-status" = continue, "floor-status" = grab and "scope" = any.

The configuration of the dialogue system allows the level of acknowledgment to be specified so that indicating "agree" as the level to acknowledge will cause the system to only issue acknowledgments of agreement while indicating "hear" will cause the system to issue acknowledgments whenever it "hears", "understands" or "agrees". If "hear" is the level set and an acknowledgment of "hear", "understand" and "agree" are intended then only the highest level acknowledgment type

of “agree” will be issued since it presupposes the other lower levels (e.g. can’t agree or disagree if didn’t understand and can’t understand if didn’t hear properly because of background noise).

*PWJ 8/25/05: can student utterances be matched against transitions definitions as well as concept definitions and the attribute values for where found returned?* **MH 7/19/2007: (UP-DATED)** This is currently not implemented as the transition definitions are maintained by the NLG module and there is no mechanism in place to translate them into concept definitions. The Dialogue Manager would also have to know when and whether to augment the language model it sends to NLU, or alternatively, understand when a transition concept is returned even though it is not in the model (in the same way unanticipated response is handled). The issue is largely one of representation.

### 3.4.1 Transitions in *sc*

Transition chunks are purely optional. TuTalk has a set of built-in transitions for English.

The “override” keyword causes the NLG module to replace the phrases for this transition type in its built-in transition table (if it has one). The default behavior is to supplement any existing acknowledgments with those provided in the scenario.

**Example:**

```
1 trans agree pos override
2   "By golly, I agree wholeheartedly!"
3   "Jolly good" lang en-GB
```

## 3.5 Module Configuration

The TuTalk Coordinator knows which modules it needs to launch, and modules know their default settings. Thus, this section is for overriding those defaults, for providing additional configuration data for submodules (**FIXME: documented somewhere, someday, in some part of this document**), and – in very rare cases – for completely replacing a module.

The recommended way of providing an alternative module that takes additional or different parameters, is not to replace it, but rather to implement a submodule API that allows overriding of the main functionality.

It is also possible to provide additional modules beyond the standard ones. A `<module>` element can provide a `file` attribute. `<module name='nlu' file='CustomNLU.py' />` would cause “CustomNLU.py” to be executed, instead of the standard “NLU.py”. The approach of replacing modules is, the reader should note, discouraged. Replacement modules written in a language other than Python are doubly so.

### 3.5.1 Configuration in *sc*

A module configuration option starts with “config”. Next is the short name of the module from or “global”, to define attributes for the scenario as a whole these are the `<scenario>` attributes from `TuTalkScenario.dtd`.

What follows is a series of key-value pairs, as necessary to customize module behavior. Module customization options are discussed in the relevant modules’ documentation below (or is it above??). Allowed `<scenario>` attributes are the optional `description` and `version`, and the required `default-language`, which in `sc` defaults to “en” if not supplied.

To specify a more complex structure like an array, or a nested sequence of objects, you have to use a Python initializer in quotes, beginning with the 'Python' keyword. Fortunately, this is rarely necessary. Currently it is used when the Normalizer lexicon must be extended because the scenario uses domain-specific words or proper nouns not in the supplied lexicon(s). A dictionary looks like this:

```
1 "Python{'key1':'value1', 'WDUQ':90.5}"
```

An array looks like this:

```
1 "Python(1,2,3,4,5)" or "Python[1,2,3,4,5]"
```

Any line in a config chunk can start with 'config'. It's optional at the beginning of each line after the first. That makes it easier to comment/uncomment config lines.

**Example:**

```
1 # As mentioned above, it is not necessary to specify the default language
2 config global version "$Revision: 1.7 $" default-language en
3 # We use several proper nouns in the scenario, so we declare them here,
4 # and provide morphological info so en-normalizer.py won't spell-correct them.
5 # A different normalizer will probably require (and hopefully document)
6 # a different format.
7 config normalizer lexicon-supplement "Python{'en':
8   ['springfield n-springfield-',
9     'chicago n-chicago-',
10    'ontario n-ontario-',
11    'illinois n-illinois-',
12    'superior n-superior-',
13    'huron n-huron-',
14    'erie n-erie-',
15    'michigan n-michigan-',
16    'ontario n-ontario-'
17   ]}"
18 # Aren't you glad quoted material can span several lines!
```

### 3.6 The Syntax of sc

In the quasi-BNF form used here, we do not indicate whitespace used to separate line elements (keys-value pairs and the like). We also use regular expression-style "\*" and "+" for types of repetition ( $\geq 0$  and  $\geq 1$  iteration, respectively).

We also ignore comments, which `sc` in fact strips out of the document before it is parsed.

Here are some top-level definitions, which are whitespace-sensitive:

```
1 LITERAL      ::= ''' <TEXT> '''
2 SLITERAL     ::= ''' <TEXT> [ | <TEXT> ] * '''
3 QLITERAL     ::= [ " ] <TEXT> [ " ]
4 TOKEN        ::= ( <LETTER> | <DIGIT> | "." | "-" | "_" | ":" ) +
5 QTOKEN       ::= [ " ] ( <LETTER> | <DIGIT> | "." | "-" | "_" | ":" ) + [ " ]
```

```

6  TOKENS      ::= ''' <WHITESPACE>* TOKEN (<WHITESPACE>+ TOKEN)* <WHITESPACE>* '''
7  PYTHON      ::= 'Python' <PYTHON-INITIALIZER> '''

```

Quoted data (like LITERAL) can contain newlines, which are discarded during compilation. Tokens in angle brackets are not defined here, but their meanings should be fairly obvious. Elements that begin with “Q” have optional quotation marks.

Within quoted text, double quotes may be escaped with a backslash. Within unquoted text, a line can be continued by placing a backslash at the end of the line.

```

1  SC-FILE      ::= <WHITESPACE>* CHUNK (<EOL> <EOL>+ CHUNK)* <WHITESPACE>*
2  CHUNK        ::= CONFIG-CHUNK | TRANS-CHUNK | CONCEPT-CHUNK | GOAL-CHUNK
3
4  CONFIG-CHUNK ::= CONFIG-TOP (<EOL> CONFIG-LINE)*
5  CONFIG-TOP   ::= "config" MODULE-NAME (CONFIG-KEY CONFIG-VALUE)*
6  CONFIG-LINE  ::= ["config"] MODULE-NAME (CONFIG-KEY CONFIG-VALUE)*
7  MODULE-NAME  ::= "dh" | "dm" | "in" | "nlg" | "nlu" | "normalizer" | "out" |
8               "session" | "student" | "global"
9  CONFIG-KEY   ::= TOKEN
10 CONFIG-VALUE ::= TOKEN | TOKENS | PYTHON
11
12 TRANS-CHUNK  ::= TRANS-TOP (<EOL> TRANS-LINE)*
13 TRANS-TOP    ::= "t[rans]" (ACK-TYPE | ACK-POLARITY | TOPIC-STATUS |
14                  FLOOR-STATUS | SCOPE | "override")*
15 TRANS-LINE   ::= LITERAL ["lang[uage]" <ISO-LANGUAGE-CODE>]
16 ACK-TYPE     ::= "agree" | "understand" | "hear" | "none"
17 ACK-POLARITY ::= "pos" | "neg" | "mixed"
18 TOPIC-STATUS ::= "new" | "interrupt" | "repeat" | "refresh" | "resume" |
19               "finish" | "continue"
20 FLOOR-STATUS ::= "concede" | "grab" | "neutral"
21 SCOPE        ::= "immediate" | "nonimmediate" | "any"
22
23 CONCEPT-CHUNK ::= C-TOP (<EOL> C-LINE)*
24 C-TOP         ::= "c" (C-NAME ["transition"])*
25 C-NAME        ::= TOKEN
26 C-LINE        ::= LITERAL (C-DIFF | C-LANG | C-FLAG | "min[imal"])*
27 C-DIFF        ::= "diff[iculty]" TOKEN
28 C-LANG        ::= "lang[uage]" <ISO-LANGUAGE-CODE>
29 C-FLAG        ::= "flag" TOKEN
30
31 GOAL-CHUNK    ::= G-TOP [<EOL> G-COMMENT] (<EOL> G-STEP)+
32 G-TOP         ::= "g" G-NAME (G-SEM | G-LOOP | G-DIFF)*
33 G-NAME        ::= TOKEN
34 G-SEM         ::= "sem" QTOKEN
35 G-LOOP        ::= "loop" (TOKEN | TOKENS)
36 G-DIFF        ::= "diff[iculty]" TOKEN
37 G-COMMENT     ::= ("% " | "comm[ent]") TEXT (<EOL> ("% " | "comm[ent]") TEXT)*

```

```

38 G-STEP          ::= (G-SUBGOAL | (G-INITIATION [G-RESPONSE* G-ELSE])) G-OPT*
39 G-SUBGOAL       ::= ("do" | "subgoal") G-NAME
40 G-INITIATION    ::= ("say" | "initiation") (C-NAME | SLITERAL) G-INIT-ATTRS
41 G-INIT-ATTRS   ::= (G-SEM | G-TRUTH | G-KC)*
42 G-TRUTH         ::= "true" | "false" | "partial"
43 G-KC           ::= "kc" QLITERAL
44 G-GOALNAMES     ::= TOKENS
45 G-RESPONSE      ::= ("if" | "response") (C-NAME | SLITERAL) G-RESP-ATTRS
46 G-RESP-ATTRS   ::= (G-SEM | G-OPT | G-TRUTH | G-SAY | G-DO | G-KC)*
47 G-ANSWER        ::= "answer" (C-NAME | LITERAL)
48 G-OPT           ::= "once" | "rand[om]" | "opt[ional]"
49 G-SAY           ::= ("say" | "say-nomatch")(C-NAME | SLITERAL)
50 G-DO            ::= ("do" | "push" | "do-nomatch") G-GOALNAMES
51 G-ELSE          ::= ("else" | "otherwise") G-RESP-ATTRS

```

## 4 Dialogue Policies

A number of default policies are provided in the TuTalk deliverable. Replacement policies can be defined but are not the responsibility of the TuTalk developers. In this section we review each of the policies mentioned in the preceding sections and provide more detailed specifications for them.

## 5 The Coordinator

The Coordinator is the central unifying process in TuTalk. This is the “program” that is executed by a user.

When run, the Coordinator reads and parses the scenario it is to use, and then loads all the needed modules.

The Coordinator can be run as an interactive program or as a daemon. When run as a daemon, it redirects all of its I/O to files; the only way to communicate with it is through `tutalkd`.

The Coordinator is implemented as four threads:

1. listener waits for messages from `tutalkd` and puts them in a queue
2. writer waits for messages to be put in a queue, and sends them to `tutalkd`
3. worker handles the queued up by the listener thread
4. console accepts debugging commands from the user/developer, or when running as a daemon, just sleeps most of the time

The work thread is where all the real work is done, and because the limitation imposed by SQLite3 that a database connection not be shared across threads, any other thread in TuTalk that needs to do something that might ping the Dialogue History module, must go through the work thread. See the Input Manager source code for an example of using a `TuTalkCommand` to cause the work thread to call a function. This discipline is very desirable anyway, even when SQLite is not affected, as having a single thread responsible for most of the system’s state reduces the chance of race condition bugs, always a danger in a multithreaded program.

## 5.1 Usage

TuTalk.py [OPTIONS] [scenarioFile]

Both XML and sc files can be read. If the scenario file name is omitted, the Coordinator loads the simple TestScenario.xml file used for development.

## 5.2 Command-Line Switches

This is a summary of the Coordinator command-line usage screen.

Usage: TuTalk.py [OPTIONS] [SCENARIO-FILE]

Run the TuTalk Coordinator on SCENARIO-FILE, or TestScenario.xml if omitted.

-c, --console	All debugging info goes to console as well as logfile.
-d, --daemon	Run as a daemon.
--delete-history	Remove the Dialogue History database the tables 'history', 'groups', and 'globals'. Prompts for confirmation if not run in daemon mode.
-e STR, --exec=STR	Execute the command STR (such as 'fd' or 'modules') when all modules are ready. STR must be quoted if it contains spaces.
-h, --help	Print this summary and exit.
-p, --profile	Run the Python profiler on TuTalk.
-r N, --random=N	Seed the random number generator used by various modules to the integer value N. This is for replaying log files.
-u USER, --user=USER	Run as an experimenter named USER.
-v, --verbose	Emit debugging info; repeat for more verbosity.

## 5.3 TuTalk Commands

At the terminal, the Coordinator accepts several commands that may be useful in debugging.

- `debug <module> [payload]` asks a module to report its internal state. The module is identified by its “short name” – for example, the Dialogue Manager Module’s short name is `dm`. Any trailing input after `debug` is sent to the module and (typically) `eval()`ed as Python code. This is hard to write but may be useful for debugging a module if it acts strangely.
- `exec payload` is, like `debug`, for debugging, but it differs in two ways. First, it only `eval()`s code within the Coordinator rather than sending it to a module. Second, it requires a payload to `eval()`.
- `sim [payload]` begins or continues a simulated student session
- `unsim` ends a simulated student session
- `fd <filename>` simulates a student by randomly picking from the expected responses for a tutor turn. If a filename is provided a copy of the dialogue is saved under that filename.

- ? prints a usage summary

## 5.4 Simulating a Student

TuTalk has the ability to log in a fake student and use the command line for a simulated conversation. When you issue the `sim` command, it begins (if necessary) a session for a special fake student ID. Any additional text (with that initial command or subsequent) is processed as input when requested by the Dialogue Manager. Tutor output is sent to the terminal and displayed in blue. *Note: once you have started a simulation (i.e., typed `sim` once), you don't have to keep typing it – any command TuTalk can't recognize is assumed to be simulated input.* Only one simulated student can be run at a time.

To log off as the simulated student, type `unsim`.

## 6 The XML Communication Format: TuTalkMessage

TuTalkMessage is the XML application used by TuTalk for Coordinator-Module and client-TuTalk communication. It is a simplified form of the NeXT/Apple Property List<sup>17</sup> (or *plist*) in its XML flavor. It is capable of serializing arrays, hashes (dictionaries), strings, real numbers, and integers to arbitrary nesting depths. It translates well to and from basic built-in Python, Java, and Objective-C types. For the remainder of this section, we will refer to XML in this format as a *message*.

The top-level document type of a message is simply `tutalk`. This node has several attributes that are mainly for routing. The most important of these are `type` and `id`. The `type` indicates what kind of service the module is requesting. In order to request that a string be normalized, a module would compose a message with the attribute `type='normalize'`.

The dictionary node contained within the `tutalk` container is generally referred to in the TuTalk source code as the `payload`. Because arbitrarily large and elaborate data structures can be embedded in it, it scales well to complex data. The payload can also be empty. It all depends on what kind of payload a given module expects for each of the services it provides.

TuTalk and `tutalkd` compose all their messages with a built-in DTD (a “standalone” document), so clients do not need to keep a local copy of it. It is safe for client applications to use the doctype line that indicates a local (to TuTalk) copy of the DTD, which indeed TuTalk has:

```
<!DOCTYPE tutalk SYSTEM 'TuTalk.dtd'>
```

The following is the current message DTD:

```

1 <!ENTITY % TuTalkObject "(array | dict | real | integer | string)?" >
2 <!ELEMENT tutalk (%TuTalkObject;)?>
3 <!ATTLIST tutalk
4   uid NMTOKEN "na"
5   gid NMTOKEN "na"
6   type NMTOKEN #REQUIRED
7   id NMTOKEN #REQUIRED
8   oneway NMTOKEN #IMPLIED
9   time NMTOKEN #IMPLIED>
10 <!ELEMENT array (%TuTalkObject;)*>
```

<sup>17</sup>[http://en.wikipedia.org/wiki/Property\\_list](http://en.wikipedia.org/wiki/Property_list)

```

11 <!ELEMENT dict (key, %TuTalkObject;)*>
12 <!ELEMENT key (#PCDATA)>
13 <!ELEMENT string (#PCDATA)>
14 <!ELEMENT real (#PCDATA)>
15 <!ELEMENT integer (#PCDATA)>

```

The `oneway` and `time` attributes are relics from an earlier version of TuTalk and are no longer used; they can be ignored but are kept in the DTD for compatibility.

The four element types supported by the NeXT/Apple `plist` DTD, that are not supported in `tutalk` DTD, are `data` (Base-64 encoded binary data), `date`, `true`, and `false`. One or more of these (`data` in particular) may be supported in the future as necessary.

## 7 tutalkd

The `tutalkd` is a lightweight server that manages and allows communication with the various scenarios that may be running on a machine. On any given secure machine, it is effectively impossible to dynamically punch holes in the firewall to allow outside access to a scenario. Since multiple scenarios can run, it is impossible to hardcode port numbers. Using port 80 with a webserver has been tried, but the CGI approach was cumbersome, as clients had to poll inefficiently in order to get tutor output – TuTalk could not “push” the information to the client, as HTTP is a connectionless protocol.

`tutalkd`, then, listens on a known port (11666) for all TuTalk-related traffic. Client traffic is routed to the scenario and server specified by the client.

Communication is full-duplex, using `\r\n` (CR+NL) as a message delimiter. The protocol for talking *to* `tutalkd` is one of simple space-separated commands or keywords, typically followed by a packet of XML as the final element. Each command type has a fixed number of parameters, but different commands have different numbers of parameters.

The first element is always a SHA-1<sup>18</sup> checksum of the remainder of the message (minus the `\r\n` delimiter which is never considered part of the command). This is to guard against man-in-the-middle attacks involving packet injection/spoofing, but it will not guard against the entire message being hijacked. The second element is always the command name. It is frequently followed by a scenario filename or full scenario name.<sup>19</sup> **NOTE: in all the discussion that follows in this section, we do not include this leading checksum in our examples, as it would be distracting. Please note that *all* traffic pointed at `tutalkd` is required to have this SHA-1 checksum. We also do not include the delimiter, `\r\n`, in our examples (nor, indeed, include them in SHA-1 calculations). For example:**

When our example is:

```
launch TestScenario.xml moses
```

We actually send:

```
b7a2158bc691b678336a90faa2100c8c7b3dabfe launch TestScenario.xml moses\r\n
```

<sup>18</sup><http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>

<sup>19</sup>“Full scenario name” is the basename of the lockfile created by the Coordinator: if the scenario filename is “Physics.xml” and it is being run by the user “pam”, then the full scenario name is “TuTalk-Physics-pam”.

`tutalkd` always sends an XML response to a command when the command is a request for information from `tutalkd` (specifically, the `time`, `auth`, and `coordinator` commands). When the command is for a TuTalk scenario, `tutalkd` returns an empty string (i.e., just the `return-newline` sequence), as it is expected that TuTalk itself will provide a response if one is appropriate.<sup>20</sup>

A `tutalkd` XML response is always an XML TuTalk Message, with a `type` header attribute the same as the original command type. When `tutalkd` generates an XML reply, typically the reply has a single key in the payload dictionary: `ok` or `error`. In the case of `ok`, the associated string value will contain whatever information was requested from `tutalkd`. For example, on successful completion of the `auth` command, the value is the UUID used to create a session key. However, in the case of the `scenarios` command, the payload does not contain `ok`, but rather each key is the name of a scenario. This is a special case because the return value is a list rather than a single datum. In the case of `error`, the value associated with that key is an error string, often with Pythonic backtrace or Exception information, in case `tutalkd` has gotten *really* confused.

There are two “privilege levels” in `tutalkd` commands. The “normal” level is for simple clients. Clients need only be able to log in, send input to the tutor, and log out. Clients need no authentication to talk to `tutalkd`. Any authentication will be handled by the Session Manager or other parts of TuTalk.

The higher – “privileged” – level is for experimenters and authoring tools. Privileged users must use the `auth` and `privileged-login` commands to establish identity.

Because clients – privileged or not – need to issue TuTalk Messages to running scenarios, `tutalkd` injects into the XML payload a key-value pair that indicates whether or not the client is privileged. Whenever a client calls the `submit` command, this security-related information is provided so that modules can decide whether to honor a service request, and how to respond to failure. This enables clients to test a module – and perhaps even manipulate its state – in a live system. The module can always refuse to allow certain operations to some privilege classes should they be potentially destabilizing.

The following are the commands recognized by `tutalkd`.

Who sends	Command	Parameter 1	Parameter 2	Parameter 3
TuTalk	<code>register</code>	scenario	module name	-
TuTalk	<code>send</code>	scenario	uid	XML (e.g. <code>tutor-turn</code> )
Client	<code>login</code>	scenario	uid	XML <code>login</code>
Client	<code>submit</code>	scenario	module name	XML (e.g. <code>input</code> )
Client	<code>time</code>	-	-	-
Privileged	<code>auth</code>	uid	-	-
Privileged	<code>privileged-login</code>	uid	password hash	-
Privileged	<code>scenarios</code>	-	-	-
Privileged	<code>upload</code>	filename	uid	file contents
Privileged	<code>launch</code>	filename	uid	-
Privileged	<code>coordinator</code>	scenario	uid	command (e.g. ‘ <code>!!!quit</code> ’)
Privileged	<code>account</code>	uid	encrypted user info	-

<sup>20</sup>The empty string is for the benefit of clients that synchronously call `recv()` on the socket, waiting for a response. Without a response, the thread or program can block indefinitely waiting for a response that will never arrive.

## 7.1 The register command

A TuTalk module uses this to establish a persistent socket-based connection with `tutalkd`, and a unique name to identify itself to `tutalkd` and clients. In our default setup, two modules register: Input and Output. They identify themselves by the names `input` and `output`, respectively. A client needs to know the name of the appropriate server when it issues a `submit` command (see below). Typically this will be `input`.

*It is always possible to provide servers other than `input` for the client to submit information to. If you are writing a Wizard of Oz interface so that a human can provide the NLU service, as is being done in the ITR project, then you may have a stub NLU module that registers under the name `nlu`, and sends and receives `nlu` requests with a special client.*

**Example:**

```
1 register TuTalk-TestScenario-moses input
```

## 7.2 The send command

TuTalk modules use this to send XML data to a single user. If `tutalkd` cannot locate the specified user in the current scenario, it tries to find a privileged user of the same name. If it can find neither, it returns an error message. Otherwise it simply forwards the XML to the client socket.

The prototypical example would be the `tutor-turn` message:

**Example:**

```
1 send TuTalk-TestScenario-moses my-name
2 (long xml header with DTD omitted...)
3 <tutalk uid="me" gid="group-me" type="tutor-turn" id="out-7189-5">
4   <dict>
5     <key>lang</key> <string>en</string>
6     <key>uuid</key> <string>f1d3d374-4f96-4b98-8499-7e15502d6f08</string>
7     <key>text</key> <string>Hello.</string>
8     <key>type</key> <string>tutor-turn</string>
9     <key>n</key> <integer>1</integer>
10    <key>of</key> <integer>1</integer>
11  </dict>
12 </tutalk>
```

Typically the client only cares about the `text` field of the `tutor-turn` type of message.

## 7.3 The login command

A client sends this command to log into both `tutalkd` and a TuTalk scenario. The client provides the target scenario, a unique name for the student, and an XML message that contains the additional data that the Session Manager uses. If the scenario name is known, and if the user name is not a duplicate within that scenario, `tutalkd` forwards the XML to `input`. Otherwise it returns an error message.

### Example:

```
1 login TuTalk-TestScenario-moses my-name
2 <?xml version='1.0' encoding='UTF-8'?>
3 <!DOCTYPE tutalk SYSTEM 'TuTalk.dtd'>
4 <tutalk uid='my-name' type='login' id='java-0-0'>
5   <dict>
6     <key>agent-type</key> <string>student</string>
7     <key>provides</key>
8       <array><string>tutor-turn</string><string>login-response</string></array>
9   </dict></tutalk>
```

## 7.4 The submit command

Clients use this to submit their student turns. The client provides the scenario name and the server (typically `input`) that will be sent the XML. The most common `submit` case will usually be an `input` type XML request sent to the `input` server.

### Example:

```
1 submit TuTalk-TestScenario-moses input
2 <?xml version='1.0' encoding='UTF-8'?>
3 <!DOCTYPE tutalk SYSTEM 'TuTalk.dtd'>
4 <tutalk uid='my-name' gid='group-my-name' type='input' id='cgi-0-0'>
5   <dict>
6     <key>text</key>
7     <string>Can you hear me now?</string>
8   </dict>
9 </tutalk>
```

## 7.5 The time command

This command takes no parameters. It retrieves from `tutalkd` the time according to the machine it is running on. It returns a payload in which the `ok` key points to a real number in Unix time<sup>21</sup>. This command is useful for logging in client programs, in which timestamps can be synchronized with TuTalk's database and logging timestamps.

### Example:

```
1 time
```

The response from `tutalkd` might look like this...

---

<sup>21</sup>Seconds since UTC midnight, 01/01/1970

```
1 <tutalk type='time' uid='na' gid='na' id='tutalkd-0-0'>
2   <dict>
3     <key>ok</key>   <real>1169252062.173027</real>
4   </dict>
5 </tutalk>
```

## 7.6 The auth command

This command takes only one parameter: the user ID of the experimenter. The command is used as the first step in authorizing an experimenter. In response, `tutalkd` sends a TuTalk Message of type `auth`. Its payload is a dictionary with only one field: named either `ok` or `error`. If `ok`, then the value is a UUID string that is to be used as a session key. If `error`, then the value will be a human-readable explanation of the error.

### Example:

```
1 auth moses
```

The response from `tutalkd` might look like this...

```
1 <tutalk type='uuid' uid='na' gid='na' id='tutalkd-0-0'>
2   <dict>
3     <key>ok</key>   <string>02e4e485-4f96-4b98-8499-7e15502d6f08</string>
4   </dict>
5 </tutalk>
```

Or this...<sup>22</sup>

```
1 <tutalk type='uuid' uid='na' gid='na' id='tutalkd-0-0'>
2   <dict>
3     <key>error</key> <string>connection is already privileged</string>
4   </dict>
5 </tutalk>
```

## 7.7 The privileged-login command

This command takes the user ID of the experimenter and a hash of the user's password and the UUID returned by the previous `auth` command. More specifically, the hex string representation of SHA-1 digest of the experimenter's password is taken, and the UUID is concatenated onto it. A second SHA-1 hash is made of this composite string, and its stringified (hex) form is sent as the second parameter to this command.

---

<sup>22</sup>In the case of this particular error – “connection is already privileged” – it means the experimenter has already checked out OK with a different session key. To `tutalkd`, it looks like an imposter is trying to pose as this experimenter.

In Pseudocode:

```
pwhash = SHA1(SHA1(password) + uuid)
```

In Python:

```
pwhash = sha.new(sha.new(password).hexdigest() + uuid).hexdigest()
```

**Example:** The user name is “moses” and the password is “blah”.  
tutalkd has returned 1b9597cd-d44f-495b-8df0-963ce8888f0a  
as the UUID session key.

```
SHA1('blah') = '5bf1fd927dfb8679496a2e6cf00cbe50c1c87145'
```

```
SHA1('5bf1fd927dfb8679496a2e6cf00cbe50c1c871451b9597cd-d44f-495b-8df0-963ce8888f0a') =  
'0d87175b3d1493d1aeb4d76b6c77086e31c687d3'
```

```
1 privileged-login moses 0d87175b3d1493d1aeb4d76b6c77086e31c687d3
```

There will only be a response if there is an error. Error messages from security-related commands such as this are deliberately cryptic in an attempt to discourage script kiddies.<sup>23</sup>

Once the experimenter is elevated to a privileged state, any privileged command may be issued for a scenario he owns. (Only the `root` user, if set up, has unrestricted access to all scenarios.) When the connection is closed, the privilege expires. Hence the use of the term “session key” for the UUID. By “closed” we mean the socket is closed. In the case of a CGI script (like `admin.cgi`) which can access privileged services, it must call the `auth`, `privileged-login` sequence each time it executes before it can successfully issue a privileged command.

## 7.8 The scenarios command

This command takes no parameters at all. In response, `tutalkd` sends a TuTalk Message of type `scenarios`. It reports, for each full scenario name, the uid’s of logged-in students, and the scenario status as reported by the coordinator:

**Example:**

```
1 scenarios
```

`tutalkd` will return a message that looks like this:

```
1 (long xml header with DTD omitted...)  
2 <tutalk uid='na' gid='na' type='scenarios' id='tutalkd-0-0'>  
3 <dict>  
4 <key>TuTalk-TestScenario-moses</key>  
5 <dict>  
6 <key>students</key>  
7 <array>
```

<sup>23</sup>“Not authorized” as an error message may mean no previous `auth` command was made for this socket connection, or it may mean the password is wrong. We don’t give away what actually happened (except possibly in the log files).

```

8     <string>student-1</string>
9     <string>student-2</string>
10    <string>student-3</string>
11    <array>
12    <key>status</key> <string>Running</string>
13  </dict>
14 </dict>
15 </tutalk>

```

## 7.9 The upload command

This command is used to upload a scenario file into an area owned by the privileged experimenter: `TuTalk/Experimenters/experimenter-name/scenarios/`. The first parameter is the name of the file to be created on the server where `tutalkd` is running. The second is the experimenter uid. The final parameter is the content of the scenario file.

This command returns the standard `ok` or `error` dictionary.

`tutalkd` runs the `xmllint` facility to validate the XML file once transferred. If validation fails, the file is removed. Ill-formed scenario files are not allowed to pass through, as it is typically easier to diagnose an XML problem using a specialized tool than by attempting a launch using TuTalk and having to inspect the Coordinator logs manually. `xmllint`'s diagnosis of the problem is returned as the `error` value in case validation fails.

### Example:

```

1 upload TestScenario.xml moses <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE scenario SYSTEM "http://www.pitt.edu/~TuTalk/TuTalkScenario.dtd">
3 (many more lines of XML...)

```

## 7.10 The launch command

This command is used to start an instance of the TuTalk backend for a given scenario file, which is given in the first parameter. This file is understood to live in the directory: `TuTalk/Experimenters/experimenter-name/scenarios/`, which is where the `upload` command places files.

The second parameter is the experimenter uid in whose name the TuTalk scenario is to be launched. **Note:** *only the “root” user, if one exists, can launch a scenario as any other user.*

### Example:

```

1 launch TestScenario.xml moses

```

## 7.11 The coordinator command

This command is used to send commands directly to the Coordinator of a given scenario. Coordinator-direct commands always have the sequence `“!!!”` prepended. There are various Coordinator commands, but those most likely to be of interest are `“!!!status”` and `“!!!quit”`.

The first parameter is the full scenario name. The second is the experimenter uid. **Note:** *only the “root” user, if one exists, can issue commands to a scenario it does not own.* The third parameter is the “!!!” command string.

This command returns an **ok** or **error** dictionary. Any response from the Coordinator is returned as the value for the **ok** key. This response can be subject to interpretation. The **!!!modules** command returns a stringified Python structure, whereas the **!!!debug module-name** command returns nothing as debugging information is sent to standard output (and hence into a log when TuTalk is running as a daemon process).

**Example:** (called by “root” to reload a scenario uploaded and launched by the experimenter “moses”. A rather sneaky thing to do. That’s why only “root” can do this kind of stuff.)

```
1 coordinator TuTalk-TestScenario-moses root !!!reload
```

## 7.12 The account command

This command is for adding or updating user records in `tutalkd`’s “Authdata” file. The first parameter is the submitter’s user id.

The second parameter is an encrypted 5-field `/etc/passwd`-style user info record:  
**uid:name:mail:phone:password-hash** Record fields are separated by colons; any of them can be empty in which case the corresponding field is not updated. Literal colons inside fields must be escaped with a backslash.

The user info record is encrypted using the submitter’s password hash as a key. The cipher used is the RC4 stream cipher<sup>24</sup> in its original (somewhat insecure) form, i.e., without initially discarding any of the key stream data.

If the uid field is empty, one’s own record is updated. Otherwise a new account is created. If a user by that name already exists, `tutalkd` will report an error unless the submitter is the root user.

**Example:** (here the user is updating his own account, so the uid is not specified, and phone number and password are not being altered)

```
1 account moses 52adfa90b5cff8d5c1cccb08d94f074b7749e71480c39a0d9...
2 ...28db53b39ee970ae73c8f197308fbd2e8870a76bddef40d36
3 ...where the last parameter is an RC4-encrypted record
4 :Brian 'The Moses-meister' Hall:moses@blugs.com::
```

**FIXME:** currently multiple logins disconnecting only send a single disconnect to the Session Manager. (This probably doesn’t make sense to anyone but Moses.)

## 8 Writing a Client

Writing a client means implementing an appropriate subset of available `tutalkd` commands. Fortunately, some of the more complicated commands (e.g., `account`) should never need to be used by a client.

<sup>24</sup>Actually, it is the *alleged* RC4 cipher or ARC4. See Wikipedia for more info.

A client will typically establish a socket connection with `tutalkd` on port 11666 in a separate thread, and poll it for incoming data, splitting off complete messages when the delimiter (return+newline) is encountered. Student activity will typically be handled in the main thread, with `tutalkd` commands queued for sending by the socket listener thread, or another thread. Naturally, how this is done will depend on the implementation language. We have example clients in Java, Python, and Objective-C (Cocoa). The Java client uses a package in `client/Java/TutalkCore` that can be reused. None of the other clients, while hopefully educational for implementors, are currently suitable for wholesale code reuse.<sup>25</sup> *Note: CGI is not appropriate for writing a client, as the student will be seen to disconnect (terminating the session) every time the HTTP request completes.*

**Note:** *Our Java sample code aside, a client to be used in an experimental setting should never make the user ID visible to a student! It should be entered by an experimenter in a bulleted text field or served up by some variety of secure “uid server” as appropriate.*

## 8.1 Messages Sent by the Client

The following are, roughly in the order in which they might be sent, the messages sent to `tutalkd` for routing into the appropriate scenario. `tutalkd` commands and the protocol for sending them are covered in a previous section; we concentrate here on the appended XML data that is meant to be forwarded to a TuTalk scenario.

### 8.1.1 login

This establishes a connection such that `tutalkd` can map the client socket to a user ID, and passes the XML login data to the specified scenario. Use `tutalkd`'s `login` command to send a TuTalk Message of type `login` to TuTalk.

The `agent-type` key is used by the Session Manager to arrange users into groups. It keeps track of the number of and type of participants needed to form a session. The most common type is just a single student, so this information is optional. It defaults to `student`. Thus, line 5 in the example below is redundant. We include it only for the sake of clarity.

The `provides` key indicates which types of tutor messages the client is interested in. This gives a wizard client, for example, a way to opt out of receiving tutor text meant for the student, which would otherwise take up bandwidth and clutter the `tutalkd` logs. Currently, `tutor-turn` is the only message type for which a client can opt out. Line 8 of the example below indicates the client wishes for tutor turns.

In general, the `id` and `time` attributes in all TuTalkMessages sent by the client can be dummy values. TuTalk has fairly strict rules about how client messages are handled; appropriate values will be substituted internally if necessary.

#### Example:

```
1 login TuTalk-TestScenario-moses student1 <?xml version='1.0' encoding='UTF-8'?>
2 <!DOCTYPE tutalk SYSTEM 'TuTalk.dtd'>
3 <tutalk uid='student1' type='login' id='java-0-0'>
4 <dict>
```

---

<sup>25</sup>The Cocoa client for Mac OS X is eminently suitable for encapsulation, should single-platform clients be permissible.

```

5   <key>agent-type</key> <string>student</string>
6   <key>provides</key>
7     <array>
8       <string>tutor-turn</string>
9     </array>
10  </dict>
11 </tutalk>

```

**Note:** *It is important that the user ID sent in the `tutalkd` command is the same as the user ID sent to TuTalk in the TuTalkMessage header. If they are different, `tutalkd` will not be able to route TuTalk output to the appropriate client!*

### 8.1.2 input

Use `tutalkd`'s `submit` command to send a TuTalkMessage of type `input` to the appropriate module, typically (unless you've rewired TuTalk) the Input module, whose short name is also `input`. Send the student's input as the value for the payload `text` key.

**Example:**

```

1 submit TuTalk-TestScenario-moses input <?xml version='1.0' encoding='UTF-8'?>
2 <!DOCTYPE tutalk SYSTEM 'TuTalk.dtd'>
3 <tutalk uid='student1' type='input' id='java-0-0'>
4   <dict>
5     <key>text</key> <string>Hi, how are you?</string>
6   </dict>
7 </tutalk>

```

### 8.1.3 logout

Use `tutalkd`'s `submit` command to send a TuTalkMessage of type `logout` to the appropriate module (as above). The payload is left empty (the `<dict/>` element is an empty dictionary).

**Example:**

```

1 submit TuTalk-TestScenario-moses input <?xml version='1.0' encoding='UTF-8'?>
2 <!DOCTYPE tutalk SYSTEM 'TuTalk.dtd'>
3 <tutalk uid='student1' type='logout' id='java-0-0'>
4   <dict/>
5 </tutalk>

```

## 8.2 Messages Sent to the Client

### 8.2.1 login-response

This message indicates either login failure, or that the user may have to wait before the session can begin. The former can happen for any number of reasons; another user with the same ID may be already logged in, or the user ID could be absent from the scenario's list of permitted user IDs.

**Example:**

```

1 <tutalk type="login-response" id="o-905-1" uid="student1" gid="na">
2   <dict>
3     <key>status</key> <string>ok</string>
4     <key>text</key>   <string>You have logged in successfully,
5                         please be patient for the other
6                         necessary parties to login
7                       </string>
8   </dict>
9 </tutalk>

```

### 8.2.2 gid-assignment

Assigns a group ID to the client. Should be retained and sent as the `gid` field in subsequent TuTalk Messages. Group ID in `group-id` field of dictionary.

**Note:** *The group ID is derived from the user ID in most cases. Neither the group ID or user ID should be displayed to the user in an experimental setting!*

**Example:**

```

1 <tutalk type="gid-assignment" id="o-905-3" uid="student1" gid="group-student1">
2   <dict>
3     <key>status</key>   <string>ok</string>
4     <key>group-id</key> <string>group-student1</string>
5   </dict>
6 </tutalk>

```

### 8.2.3 tutor-turn

What the tutor says to the student. The tutor's text is the value for the `text` key in the payload.

**Example:**

```

1 <tutalk type="tutor-turn" id="o-905-5" uid="student1" gid="group-student1">
2   <dict>
3     <key>text</key>   <string>Hello.</string>
4   </dict>
5 </tutalk>

```

### 8.2.4 tutor-logout

The tutor's final message to the student. `text` may be different between the cases of student-initiated and tutor-initiated (tutor got to end of script) logout. In the example below, there is no text because the scenario had its NLG configuration set to suppress the built-in English logout acknowledgments (in this case because the scenario was not in English).

**Example:**

```

1 <tutalk type="tutor-logout" id="out-905-18" uid="student1" gid="group-student1">
2   <dict>
3     <key>text</key>   <string></string>

```

```
4 </dict>
5 </tutalk>
```

### 8.3 Detecting Errors

When `tutalkd` reports an error, it sends back response XML with a payload containing an `error` key, and an explanation as the value.

When TuTalk reports an error, it uses a `status` key with a value of `error`, and also a `text` key with the explanation as the value. This is consistent with the way in which normal tutor strings are sent.

It is done this way<sup>26</sup> because TuTalk error messages are typically more human-readable than those from `tutalkd`, which can be deliberately cryptic. `tutalkd` errors are more likely to indicate bugs in the client, whereas TuTalk errors are more likely to be “operator error” or experimental misconfiguration. Client writers may want to use different mechanisms for reporting such errors.

## 9 Modules

### 9.1 Extending TuTalk

There are two options when attempting to extend the TuTalk system with different behaviors. The first is to rewrite a module or add a new one; the second is to provide a submodule.

NLU uses matcher submodules which take a language model and a string and return a set of matches. The Normalizer uses language submodules which are in effect domain experts.

Generally, when the desired modification is to the module’s behavior where the “real work” is done, the submodule option is the easiest to code, and the easiest to reuse, as the API for a submodule is typically very small and well-defined. Only when massive changes to the overall architecture of the system are desired, should it become necessary to add or modify modules.

### 9.2 Input

This simple module processes client input that comes in via `tutalkd`.

The Dialogue Manager typically calls `requestInput()` to place a standing request for student input. An optional parameter can indicate a timeout in response to an initiative-enabled scenario which has been finished but in which the Dialogue Manager gives the student some time to resume the conversation. If the timer expires with no input, this module informs the Dialogue Manager that nothing has been said, so the session can end.

The `unrequestInput()` method cancels an existing request if it exists.

The `processInput()` method, typically called by the Coordinator, submits a client input message for processing or rejection. Any input that has not been requested by the Dialogue Manager is rejected by returning the client message with an “error” field, bearing the string “Sorry, I’m not listening right now.”

---

<sup>26</sup>Well, it was done this way by accident, but we can justify not changing it....

### 9.2.1 Configuration

The Input module does not have configuration settings currently.

*This module is not designed to accept barge-in. It only accepts input when the Dialogue Manager asks for it. In order to support barge-in, a facility for persistent input requests would need to be added, and modifications made to the Dialogue Manager so that such inputs could be handled appropriately.*

## 9.3 Normalizer

Covers spelling, stemming, punctuation markup/removal, case normalization, stopword removal, segmentation, etc. We use language-specific “experts” or “submodules”, loaded on demand. The scenario file must identify a default language, and anything in a different language must be tagged with an ISO language code (and a submodule provided for it, of course). The submodule is expected to do whatever is necessary to normalize the text; services to specifically spell-check, or stem, or whatever, are not exposed. The expert can be considered a black box.

The Normalizer provides the service `normalize`, which accepts a chunk of text (`text`) and a language key (`lang`: optional), and any other context data that may be useful to the caller when the request is completed. If the text parameter is omitted, the Normalizer sends this message on to the Input module and normalizes the student’s input when it is submitted.

```
1 <tutalk type="normalize" id="nlu-19717-3">
2 <dict>
3   <key>text</key>      <string>I don't watn ayn slaad!</string>
4   <key>lang</key>      <string>en</string>
5 </dict>
6 </tutalk>
```

On output, the `normalized` field is added. If a `lang` field was omitted, one containing the default language is added.

```
1 <tutalk type="normalize" id="nlu-19717-3">
2 <dict>
3   ...
4   <key>lang</key>      <string>en</string>
5   <key>normalized</key> <string>i do not want any salad</string>
6 </dict>
7 </tutalk>
```

### 9.3.1 Configuration

The Normalizer accepts a field called `lexicon-supplement` for supplying lexical items that are not in the supplied dictionaries for the scenario’s language(s). The value for this key is itself a dictionary of language codes mapped to arrays. Each array contains strings that the language submodule will use as lexical items. These strings are in whatever format is used by the language submodule `lexicon`.

*Example:*

```
1 <dict>
2   <key>lexicon-supplement</key>
3   <dict>
4     <key>en</key>
5     <array>
6       <string>shwoozle n-shwoozle-S</string>
7       <string>fubar adj-fubar- vlex-fubar-PDSI</string>
8     </array>
9   </dict>
10 </dict>
```

## 9.4 NLU

Out-of-the-box implementation is minimum-distance matcher. Pass it a list of concepts (a language model), and it will produce the concept label(s) that matched most closely, the text that actually matched, and its estimation of the quality of match.

Any NLU query (since it uses submodules) takes a `TuTalkMessage` as the sole argument to the `nlu()` method. This message is filled in by whatever submodules the scenario's configuration requires.

### 9.4.1 Submodules

NLU submodules provide implementations of the class `StudentModel` with the following entry points:

- `__init__(self, module, path)` where `module` is the NLU module object, and `path` is the full path to the submodule's host directory.
- `nlu(self, tutalk, concepts)` where `tutalk` is a `TuTalkMessage` of type `nlu`; see below for an explanation of its payload structure. `concepts` is a data structure containing all the concepts defined by the scenario. There is no return value: `tutalk`'s payload is modified in place.

### 9.4.2 The `nlu` Query

This is the main capability of NLU and its submodules. The payload contains a field `normalized` which is the normalized student string that is to be matched against the relevant concepts. These concepts – the language model – are contained in an array `models`, which for historical reasons is an array containing one item: a space-delimited list of concept labels.<sup>27</sup>

NLU queries its submodules in configuration order. Each is expected to inject two items into the payload when it is finished: `concepts`, a dictionary of concept labels to the matched substring, and

---

<sup>27</sup>In our original specification we anticipated multi-tiered language models to support student initiative. The idea was to have these models checked in order of appearance, with the assumption that subsequent items would be more permissive (i.e., contain more concept labels). It turned out simpler to have the Dialogue Manager simply make two requests, but the data structure was preserved for the sake of compatibility.

quality, a real number in the range of 0.0 to 1.0 inclusive. NLU renames these items by prefixing the submodule name, so there will be no naming collisions. Thus subsequent submodules have access to the previous submodule results.<sup>28</sup> The module's configuration (see below) determines whether all submodules are queried, or whether the querying terminates as soon as a result is over threshold.

Here is the input to the `nlu()` method (in module or submodule):

```
<tutalk type="nlu" id="dm-19717-3">
<dict>
  <key>normalized</key> <string>i not want any salad</string>
  <key>models</key>
  <array>
    <string>enthuse-about-appetizers skip-appetizer</string>
  </array>
  <key>uuid</key>      <string>99af1690-b4cb-479d-a3cb-767b056a29f9</string>
</tutalk>
```

On output, the message looks like this. NLU has promoted the highest quality submodule results back to `concepts` and `quality` and removed the others.

```
<tutalk type="nlu" id="dm-19717-3">
<dict>
  <key>concepts</key>
  <dict>
    <key>skip-appetizer</key> <string>I do not want</string>
  </dict>
  <key>quality</key>      <real>0.23</real>
  <key>normalized</key> <string>i not want any salad</string>
  <key>models</key>
  <array>
    <string>enthuse-about-appetizers skip-appetizer</string>
  </array>
  <key>uuid</key>      <string>99af1690-b4cb-479d-a3cb-767b056a29f9</string>
</dict>
</tutalk>
```

### 9.4.3 NLU Wizards

If an NLU submodule has a human (i.e. wizard) interface, it must not synchronously query for the interpretation as this would suspend processing for other students. Instead it should do the following:

1. Non-destructively preprocess the payload as necessary.

---

<sup>28</sup>If, for example, NLU's configuration has the submodule list `default cordillera`, when the Cordillera submodule is invoked, the payload will contain two additional fields: `default-concepts` and `default-quality`. This information can be used to guide a human interpreter.

2. raise `LibTuTalk.DeferredException` causing NLU to enqueue the message for delivery to the wizard via `tutalkd`.
3. Handle the wizard's returned message by implementing a method `returned(tutalk)` that NLU will call with the returned message.
4. Within that method, postprocess as necessary and then call the NLU module's `_continue(tutalk)` method with the message.

The file `nlu/cordillera.py` contains an example of this.

#### 9.4.4 Configuration

- **threshold** (a real number in the range 0.0 - 1.0 inclusive) Match quality required before a string can be tagged as a certain concept. *Default: 0.75*
- **submodules** (string) A space-separated list of submodules to use. The default is the built-in Levenshtein matcher. *Default: default*
- **policy** (`all`, `thresh`) Policy on how many submodules to apply to the matching task. `thresh` means bail out when a submodule is above threshold. `all` means let all submodules have a crack at it and choose the best. *Default: thresh*

## 9.5 NLG

Marks up system's dialogue response in one or more modalities. (Speech being one of the more popular alternatives.)

Given direction from the dialogue manager which comprises concepts to express and transitioning context, it will find and append together the appropriate phrases. Concepts that have the attribute "transition" will not have automatically generated transitions contiguous to them to avoid possible redundancies and conflicts.

### 9.5.1 The nlg Service

This is the main service that **NLG** provides. Its only required parameter is `concept`, a concept label. The `lang` parameter defaults to the scenario's default language, so it need not always be specified. An optional `difficulty` parameter restricts the generated text to a form of the appropriate type/difficulty for the student: essentially the caller is saying "express concept *x* in manner *y*". NLG checks with the Dialogue History Manager to see which forms of "*x* as *y*" have not been used recently, and chooses one of them randomly (falling back to those that have been seen recently if necessary). The optional `sem` tag is from the Dialogue Manager's recipe, included here so it can be sent on to the Dialogue History module.

The required `uuid`, `n`, and `of` fields are passed on the the Output module. See the Output module documentation for more information.

```
<tutalk type="nlg" id="dm-19717-13">
<dict>
  <key>uuid</key>          <string>40102a9d-5f14-4638-bdac-6ba3fc4dddb8</string>
```

```

<key>concept</key>    <string>skip-appetizer</string>
<key>lang</key>       <string>jp</string>
<key>difficulty</key> <string>ynq</string>
<key>sem</key>        <string>123</string>
<key>n</key>          <integer>2</integer>
<key>of</key>         <integer>3</integer>
</dict>
</tutalk>

```

**NLG** does not respond to this service request: it is oneway. It sends the generated text to the Output Module.

### 9.5.2 The nlg-transition Service

This service causes NLG to emit a transition or acknowledgment. The payload values are matched against the scenario’s transitions to find the closest match in terms of features. The choice is informed by the module’s `acknowledgment` config (values {`agree|understand|hear|none`}) such that, for example, a request for `ack-type=hear` and a configuration of `acknowledgment=agree` will be ignored because the module is not configured (perhaps because speech is not enabled) to provide listener feedback. (Details of algorithm forthcoming.)

**FIXME:** *Is it possible/ok to underspecify transition features like this?*

**FIXME:** *Should we use the same defaults as we have in the DTD?*

Transitions are only generated when the module configuration’s `acknowledgment` level is “higher” than the request’s `ack-type` feature (e.g. `ack-type understand` vs. `acknowledgment agree`). However, to avoid upsetting the delicate `n/of` numbering system, in case no transition text is generated, NLG still sends an empty string.

The `lang` item is optional, as usual. The scenario attributes provide a default value.

```

<tutalk type="nlg-transition" id="dm-19717-14">
<dict>
  <key>uuid</key>      <string>40102a9d-5f14-4638-bdac-6ba3fc4dddb8</string>
  <key>ack-type</key>  <string>agree</string>
  <key>ack-polarity</key> <string>pos</string>
  <key>lang</key>     <string>jp</string>
  <key>n</key>        <integer>1</integer>
  <key>of</key>       <integer>3</integer>
</dict>
</tutalk>

```

**NLG** does not respond to `nlg-transition`: it is oneway. It sends the generated text to the Output Module. Transitions are recorded separately from “normal” NLG requests; they have different `uuid` values in the Dialogue History.

The **NLG** module also responds to `set-concepts` and `set-transitions`. These are invoked by the Coordinator when the scenario is loaded, with the contents of the scenario file’s `concepts` and `transitions` sections, respectively.

### 9.5.3 Configuration

- **acknowledgment** (`agree|understand|hear|none`, default `agree`) Indicates level at which acknowledgments should be made.
  - `agree` Only acknowledge utterances that the system can assess whether it agrees or disagrees with.
  - `understand` Acknowledge that system understood (but not whether agree/disagree with that content).
  - `hear` Acknowledge whenever system believes it got a clear auditory input.
  - `none` Disable acknowledgments.
- **logout** (`yes|no`) Send logout confirmation messages (see the next three items) to the student. *Default: yes. Note: if the scenario language is not English, the following three attributes should probably be specified in the target language.*
- **finished-logout** (String) The message TuTalk sends the student when the session is finished. *Default: “This session is now finished. I have logged you out.”*
- **acknowledge-logout** (String) The message TuTalk sends to acknowledge student-initiated logout. *Default: “OK, you’re logged out!”*
- **other-logout** (String) The message TuTalk sends to indicate the student has been logged out because another (human) participant in the session has logged out. *Default: “A participant in this session logged out, so I have logged you out.”*

## 9.6 Output

This is the interface layer between NLG and the outside world. The Output Module keeps a persistent connection with `tutalkd`, using it to push completed messages to the client.

### 9.6.1 The output Service

There are two classes of TuTalk messages that Output deals with: the ones it receives from NLG and the ones it emits. The messages received from NLG are for the `output` service. What it emits (sends to a client via `tutalkd`) is based on one or more `output` requests, with some minor changes that will be discussed below.

The required `text` and `type` portions of the message payload are the content and class of output, respectively.

*FIXME: insert something here about group IDs, group maps, etc.*

The optional `uuid` is the key to a Dialogue History entry. When the student interface fetches this utterance, this module sends a timestamp to the Dialogue History Manager. The only messages that should not have a `uuid` are perhaps greetings, logoff messages, or other administrative text that may be sent to students but not *per se* part of the scenario script. This also applies to debug messages dispatched through Input and delivered to Output.

Since a single tutor turn may span multiple `output` messages, Output is required to maintain “mailboxes” in which incomplete turns are staged. The optional `n` and `of` fields encode the 1-based

message index and total number of messages, respectively. Together they indicate that the message below is “2 out of 3” and that the first and third messages have already arrived or will arrive shortly. No outgoing message is sent until all parts have been assembled. If either **n** or **of** numbers are absent, the module treats the output request as standalone.

The optional **status** field gives some semantics to the type of response being given. This is passed along to the outside UI so it can maintain the appropriate state. Currently available status tags are **ok** and **error**. Other stati may be added in the future. If **status** is not given, it is assumed to be **ok**.

```
1 <tutalk type="output" id="nlg-19727-14">
2 <dict>
3   <key>text</key>    <string>Let's review the principles involved...</string>
4   <key>type</key>    <string>tutor-turn</string>
5   <key>status</key>  <string>ok</string>
6   <key>uuid</key>    <string>ae5b2dcd-8239-44cd-a16a-8be26042ce79</string>
7   <key>n</key>       <integer>2</integer>
8   <key>of</key>     <integer>3</integer>
9 </dict>
10 </tutalk>
```

## 9.6.2 What Output Sends to tutalkd

Once all the parts of a tutor turn are assembled, they are merged into a TuTalk Message that is quite similar to its constituents. The required **type** field of the payload is promoted to the **type** field of the message header. This makes it easier for the client to dispatch incoming messages without needing to inspect the message payload.

## 9.6.3 Configuration

There are no configuration flags defined for The Output Module at this time.

## 9.7 Student Model

Called by the dialogue manager typically, to determine “how well” a student did on some particular content. Maintains policy about how it selects among alternatives, and how repetitive to be.

All Student Model queries (since it uses submodules) take a **TuTalkMessage** as the sole argument to the **query()** method. The result (typically “yes/no”) will be the value for a key **result** in the payload. However, if the query type is unknown to whatever submodules are used by the scenario, the **result** key might be absent, in which case the caller needs to supply a default value or react accordingly.

### 9.7.1 Submodules

Student Model submodules provide implementations of the class **StudentModel** with the following entry points:

- `__init__(self, module, path)` where `module` is the Student Model module object, and `path` is the full path to the submodule's host directory.
- `query(self, tutalk)` where `tutalk` is a `TuTalkMessage` (see below for types).

### 9.7.2 The skip-optional Query

Given a step's `sem` label and its optionality, determines whether or not to skip the step.

```

1 <tutalk type="skip-optional" id="dm-19717-3">
2 <dict>
3   <key>sem</key>           <string>enthuse-about-appetizers</string>
4   <key>opt</key>          <string>said-once</string>
5 </tutalk>
```

On output...

```

1 <tutalk type="skip-optional" id="dm-19717-3">
2 <dict>
3   <key>sem</key>           <string>enthuse-about-appetizers</string>
4   <key>opt</key>          <string>said-once</string>
5   <key>result</key>       <string>yes</string>
6 </tutalk>
```

When to skip an optional step:

1. if `opt="yes"`, if the initiation `sem` is recent, or if the initiation has ever appeared in case `config[optional-step-policy]` is `'once'`. Coverage may also be calculated.
2. if `opt='said-once'`, if the initiation has ever appeared. This overrides the module config, and no coverage check is ever done.
3. if `opt='random'`, the step will be skipped if our die roll is less than or equal to the value in `configuration[random-threshold]`. This overrides the module config, and no coverage check is ever done.

*FIXME: should `said-once/once` apply to mixed initiative, i.e., should we look in tutor **and** student utterances for that `sem` tag?*

### 9.7.3 The sem-covered Query

Given one or more `sem` labels (whitespace-separated), returns whether *all* of the labels should be considered covered in the dialogue history.

```

1 <tutalk type="sem-covered" id="dm-19717-4">
2 <dict>
3   <key>sem</key>           <string>sem1 sem2 sem3</string>
4 </tutalk>
```

On output...

```
1 <tutalk type="sem-covered" id="dm-19717-4">
2 <dict>
3   <key>sem</key>       <string>sem1 sem2 sem3</string>
4   <key>result</key>   <string>yes</string>
5 </tutalk>
```

### 9.7.4 The choose-goal-version Query

Given a goal name and a set of `sem` labels (whitespace-separated), returns the `sem` label of the most appropriate recipe for the goal.

```
1 <tutalk type="choose-goal-version" id="dm-19717-5">
2 <dict>
3   <key>goal</key>      <string>some-goal</string>
4   <key>sem</key>      <string>sem1 sem2 sem3</string>
5 </tutalk>
```

On output...

```
1 <tutalk type="choose-goal-version" id="dm-19717-5">
2 <dict>
3   <key>goal</key>      <string>some-goal</string>
4   <key>sem</key>      <string>sem1 sem2 sem3</string>
5   <key>result</key>   <string>sem2</string>
6 </tutalk>
```

### 9.7.5 Configuration

- **selection-policy** (`first|least-difficult|most-difficult`, default `first`) Applies if there are multiple choices for a goal. `first` means pick first branch in list. `least-difficult` means pick least difficult branch first. `most-difficult` means pick most difficult branch first.
- **phrase-difficulty-rankings** (whitespace-separated list, default `ynq whq whyq howq`) A list of numbers or symbols from least to most difficult, for different ways to express a concept (typically a tutor question).
- **phrase-difficulty-rankings** (token, default `inform`) sa number or symbol to use after coverage achieved e.g. `inform`
- **recipe-difficulty-rankings** (whitespace-separated list, default `1 2 3 4 5`) A list of numbers or symbols from least to most difficult.
- **recipe-difficulty-rankings** (token, default `0`) Number or symbol to use after coverage achieved e.g. `0`

- **coverage-threshold** (token, default 0.8) Applies only for selection-policy of least and most-difficult to guide when to move up or down the scale of difficulty. Coverage means how much of a goal the student was right about. coverage-threshold helps guide selection-policy.
- **recency-threshold** (token, default 20) How many seconds should elapse before assume something is no longer in focus/memory of hearer.
- **topic-depth-threshold** (token, default 4) How many topics can push to before may cause to forget a topic.
- **optional-step-policy** (all|recency|once|coverage|none, default all) The policy for deciding when to skip a step that is marked as optional.
  - all means consider recency and coverage,
  - recency means consider just recency,
  - once means skip if it ever is in the dialogue history,
  - coverage means consider coverage only,
  - none means never skip an optional step.
- **repeat-policy** (always|once|covered, default once) Applies to retries on goals the student has already been exposed to.
  - once If branch has been done, then it should never be selected again for this user.
  - always Disregard previous usage in decision making.
  - covered Allow to repeat a branch if all other branches have been tried at least once.
- **random-threshold** (real number between 0.0 and 1.0, default 0.5) The chance that a step flagged with optional='random' will be *skipped*.

## 9.8 Dialogue Manager

Based on student input and current dialogue state, generates one or more utterances via state transitions.

Typically posts requests for NLU-classified input, and to the Student Model to determine what alternatives of a goal to use in generating its next utterance(s).

### 9.8.1 Configuration

- **accept-initiative** (yes|no) yes means the system accepts student initiative, no means that it does not. *Default: no.*
- **DNK-testing** (yes|no) yes means the system will automatically test for matches to a built-in concept DNK. no means that it does not. *Default: no.*
- **remind** (yes|no) Should allow something to be repeated to bring it back into focus. *Default: yes.*
- **test-attempted** (yes|no) The dummy 'attempted' concept (in NLU) should be tacked on to language models. *Default: no.*

## 9.9 Dialogue History Manager

Maintains the dialogue history, allowing modules to query, and also allowing modules to create entries and fill them in. The history is world-readable but only a few modules need to write to it. The Dialogue Manager and the Input module create new entries, for tutor and student turns respectively. The Dialogue Manager, NLG, and Output modules also update these entries as they are processed.<sup>29</sup>

The dialogue history fields are:

- **uuid** (String) Primary database key; RFC-4122 Universal Unique Identifier<sup>30</sup>
- **time** (Real) Unix time<sup>31</sup> of input, output, or decision
- **uid** (String) Login ID for student
- **gid** (String) Group ID for this session
- **speaker** (String) `student` or `system`
- **goal\_name** (String)
- **goal\_index** (Integer) One-based index of goals having the same name in the script, by order in file.
- **step\_type** (String) `{initiation,response,subgoal,logout}` `logout` takes an additional subtype following a colon, so there are three types: `{logout:internal,logout:student,logout:disconnect}` for tutor-initiated, student-initiated, and possibly student-initiated or caused by a network error, respectively.
- **step\_index** (Integer) One-based index of recipe step
- **phrase\_difficulty** (String) Numbers of symbols, default possible values; `ynq,whq,whyq,howq,inform`
- **recipe\_difficulty** (String) (numbers or symbols)
- **sem** (String) From script; labels of author's own choosing
- **string** (String) Unnormalized student string, or tutor string
- **normalized\_string** (String) Normalized student string
- **matched\_answer\_string** (String) A set of substrings (currently delimited with an underscore “\_”) matching concepts.
- **concepts** (String) Concept labels defined in dialogue script, space-separated
- **truth\_values** (String) Truth values (`{yes,no,partial,unknown}`) defined in dialogue script, space-separated, corresponding to values in `concepts` field
- **coverage** (Real) A real number between 0.0 and 1.0, calculated from the `truth_values` components. Reflects the student's performance on this `goal_name`. Values are graded as follows: `yes` contributes 1.0, `no` contributes 0.0, `partial` contributes 0.5, and `unknown` contributes

---

<sup>29</sup>The Output Module's only update to tutor output is to set a timestamp when the message is sent to the student. The Input Module likewise sets the time the student input was submitted to the system. The notion is to record times as close as possible to when utterances are actually made.

<sup>30</sup><http://www.ietf.org/rfc/rfc4122.txt>

<sup>31</sup>Seconds since UTC midnight, 01/01/1970

nothing – it is not used in the denominator. For example, when multiple answer parts are enabled, a `truth_values` entry containing `{partial unknown yes no}` evaluates to `coverage` of 0.5  $((0.5 + 0.0 + 1.0 + 0.0)/3.0)$ . When `truth_values` is only `unknown`, this field contains `NULL` to indicate “no coverage”.

- **obligations** (String) Required concept labels defined in dialogue script, space-separated

The Dialogue History Manager creates a database file for each scenario. Simultaneously running scenarios will not interfere with each other. Students are distinguished within a scenario database by their `uid` and `gid` fields.

TuTalk’s dialogue history database is implemented using the public-domain C-language SQLite library via a Python frontend. This service takes a SQL string with dictionary key `sql`. The `uuid` database column is the primary key; in order to update an entry, the `uuid` must be known or passed from the module that assigned it when creating an entry.<sup>32</sup>

### 9.9.1 Public API

`def history(self, sql, cols=False)` Executes the passed-in SQL query and returns either a single `result` value, or a `[result, columnNames]` value if `cols` is `True`. `result` may be empty if the query was an `UPDATE`, or if there were no matching records in the case of `SELECT`. If nonempty, `result` is an array of arrays, where the subarrays are the value tuples returned by SQLite.<sup>33</sup> These results respect `ORDER BY` constraints if present.

When `cols` is requested, the `columnNames` item is an array of column names.

`def appendHistory(self, uuid, column, value)` This is used for things like the `concepts` column, which is a space-separated list and may need to have its contents augmented rather than overwritten. Appends `value` to the text in the column called `column` and row identified by `uuid`. No return value.

### 9.9.2 Configuration

There are no specific configuration flags for the Dialogue History Manager at this time.

## 9.10 Session Manager

The Session Manager is responsible for assigning participants to groups. By default, a single participant of the “student” type is required to start a session, but an experimental setup could require something more elaborate, like two collaborating “student” and a “wizard” type who stands in for some portion of the system. The Session Manager partitions users into groups as they log in. When the scenario’s group requirement is met, the module starts a session for the recently-completed group.

When provided with a list of valid participants, it will make sure only authorized users log in.

While a group is still incomplete, participants can log out and no one will be affected. The module simply decrements the number of participants of this type. If the group is completely empty, it is deleted. Once a session has been started, however, one user logging out will end the entire

---

<sup>32</sup>The Input Module, for example, has a `uuid` field in its payload, so that the Dialogue Manager and others can fill in the rest of the fields as the student input is processed.

<sup>33</sup>If your query is `SELECT (a,b,c) FROM ...` then you will get an array of triples.

session. All other participants are notified that someone has left the group, so the session ends right away. Furthermore, for groups of more than one student, as long as the scenario is running, that group is “sticky”. The user id’s are tied to their previous group. The Session Manager plugs users back into their old group if they have one.

### 9.10.1 Public API

`def gid(self,uid)` Public method to get gid for uid. Includes both historical and current groups. Returns None if not found.

`def isUserLoggedIn(self,uid)` Returns True if the uid is mapped to a gid, and a Group by that gid exists, and there is an agent by that name logged in.

`def login(self,tutalk)` This encapsulates an attempted login by a client, and is usually created by client software and forwarded via tutalkd. The uid field of the message header is required, and the gid field may also be used in case the user was previously in a session with a particular group and the experimenter needs the new or restored session to consist of exactly the same participants.

`def logout(self,uid,type,uuid=None)` type is one of the strings in ['internal','student','disconnect']  
Handles a client’s logging out: 1) find the group based on the uid 2) get the uids of all participants 3) notify all participants if the session had started 4) remove all participants from userids dict 5) remove the group from running groups

If type is “disconnect”, it means the client did not log out gracefully but shut down their socket connection to tutalkd, whether by quitting their app, network error, meteor destroyed client, etc. In that case we don’t try to send them an acknowledgment. Plus, tutalkd already knows they’re gone (it is the one that informs us of a disconnect). Also, it should be considered normal to get a disconnect after a user has logged out, as we currently don’t have a way to tell tutalkd that the user should be cleared out of its list of connected users. Plus they may want to re-login without disconnecting.

### 9.10.2 Configuration

- **required-agent-types** A space-separated list of `agent-type:n` where `agent-type` is one of {`student`,`borg`} and `n` is the minimum number of such agents that must be logged in to start a session. For example, an entry ‘`student:2 borg:1`’ means there must be two students and one borg. *Default: student:1*
- **allowed-uids** A space-separated string of user IDs that are allowed to log in. Experimenters are urged to avoid telling students their user IDs. **Note: the *simulated-subject* “fake” uid is automatically appended to this list.** *Default: empty string means anyone can log in.*
- **send-history** (`yes|no`) If `yes`, send a dump of the student’s session dialogue history to Output when the student is logged out. *Default: no.*

## 10 Ideas on future improvements

- Improved mixed initiative handling

- Time out behavior when expecting a response so that can go on with dialogues when are of the “chat” variety. If a response timeout is exceeded then the system could go on with one of the scripted expected responses.

