

Content Planner Module for Talking Robots

Bernd Kiefer, DFKI Saarbrücken

Saarbrücken, October 28, 2011

1 Installation

Extract the given `cplanner-x.tar` file. Everything unpacked is below the top-level directory `utteranceplanner`. That directory contains three submodules, a generic visualization component (`dataviz`), an Emacs-Java connection module (`j2emacs`) and the planning module (`uttplanner`).

The directory `utteranceplanner/uttplanner` contains a startup script (`updebugger`) and also a `build.xml` file to recompile the sources from scratch. We will call this directory the *application directory* in the following text. Using the build file will also affect the `dataviz` subproject the interactive planner is using. The relevant targets are `clean`, `cleanall`, and `compile`.

To use the Emacs features, Mac users have to use AquaMacs. Plain X11 Emacs does not seem to provide a server mode.

It is also recommended to compile the Emacs lisp files in `utteranceplanner/uttplanner/etc`. Either invoke ‘`make`’ in this directory, or compile it with your favorite Emacs incarnation manually (e.g., with AquaMacs on the Mac).

2 Execution

The utterance planner is started with the following command:

```
updebugger [-e<macs> [<path-to-emacs-binary>]] [-g|-b <inputfile>]
           [-t <traceflags>] <projectfile>
```

`projectfile` is a file that contains settings, like the path to a CCG grammar, and a list of all rule files that should be loaded. The structure of the project file is described in more detail in section 3.

By default, the planner reads the project file, and the rule files specified there, and continues in an interactive console mode with rudimentary readline support. The switches `-g` and `-b` let the planner run in other modes, namely:

- batch mode, (`-b`) processing a set of logical forms that are contained in the file that is given as argument to this flag
- GUI interactive mode (`-g` command line switch)

The `-t` flag controls tracing onto the console, using a bitmask for different phases, bit 1 to trace the match phase and bit 2 to trace the application phase. Thus, `-t 3` will show

trace information for the matching as well as the application phase. The `-t` flag can be given together with `-g`. In this case, tracing to the console will be used only when the *process* button is used, and not when the GUI tracing mode itself is activated.

In combination with the `-g` flag, The `-e` flag connects the GUI to a newly started Emacs client that will show the result of rule loading, making files and errors mouse sensitive to facilitate the editing and debugging of rule files. In Unix/Linux there is usually no need to specify the path to the Emacs binary. This is different on Macs, because only AquaMacs works as emacs server, the path to the AquaMacs binary must be specified. If the Emacs connection is properly set up, the GUI trace window described below allows to jump directly to the definition of rules used during processing.

3 Project File

The project file is a plain text file that is divided into different sections. The sections are separated by lines specifying the section names, which are enclosed in square brackets, i.e., `[Settings]` for the section containing the project settings. Empty lines are skipped, and everything following a hash character (`#`) until the end of line is treated as comment and ignored.

```
# relative paths are relative to the location of this file

[Settings]
history_file = history.txt
ccg_grammar = ../../resources/grammars/openccg/grammar.xml

[Rules Stage 1]
attributes.trf
discoursemarkers.trf
infostruct.trf
transitivity.trf

[Rules Stage 2]
callfun.trf
test.trf
foo.bar
```

Figure 1: Sample Project File

Currently, the content planner distinguishes two kinds of sections, the **Settings** section and the **Rules** section, of which there may be more than one.

The **Settings** section contains key-value pairs that are separated by the equal sign (`=`). Currently, there is only one setting in effect namely the path to the corresponding CCG grammar, whose key must be `ccg_grammar`.

The section names for rules have to start with the word **Rules**, the rest before the closing bracket is taken as the stage name for the rule list that is following the section separator.

In a rule section, every non-empty line must contain the path to one rule file. Paths can be

4 GUI Mode

The screenshot shows a development environment with three main windows:

- Emacs Editor:** Displays the file `clarification-communication.lrf`. The code defines various Prolog predicates for handling clarification, including `clarification`, `clarification_failure`, and `clarification_success`. It uses SWI-Prolog syntax for logic programming.
- Trace Window:** Shows the execution of a rule. It displays the initial goal, the content of the utterance, and the internal state (e.g., `First`, `Next`, `Mod`, `Int`, `Act`, `Num`, `sg`, `Subject`, `Wh-Restr`, `Event`, `Tense`). The final output is a clarification utterance: "I am sorry".
- Status Bar:** Indicates the project is `Bot L10 (j2e-comp J2E)`.

When started in GUI mode, the application looks for the file `.cplanner` in the application directory. This file has the same syntax as the project file described above. It also has a **Settings** section to set two of the the command line arguments permanently, so that they do not have to be provided on every startup, and a **RecentFiles** section, which is maintained by the application, that contains the recently opened project files. Figure 3 shows a sample `.cplanner` file.

```
# relative paths are relative to the location of this file

[Settings]
tracing = none # none / match / action / all
emacs = /home/kiefer/Applications/AquaMacs/bin/Aquamacs

[RecentFiles]
Rules/allrules.cpj
```

Figure 3: Sample Preferences File

4.1 Main Window

The main window, shown in figure 4, consist of the input area at the top, and the input and output LF displays below it, and a status bar for status and error messages at the bottom. The function of the buttons in the tool bar and the menu items will be described in what follows.

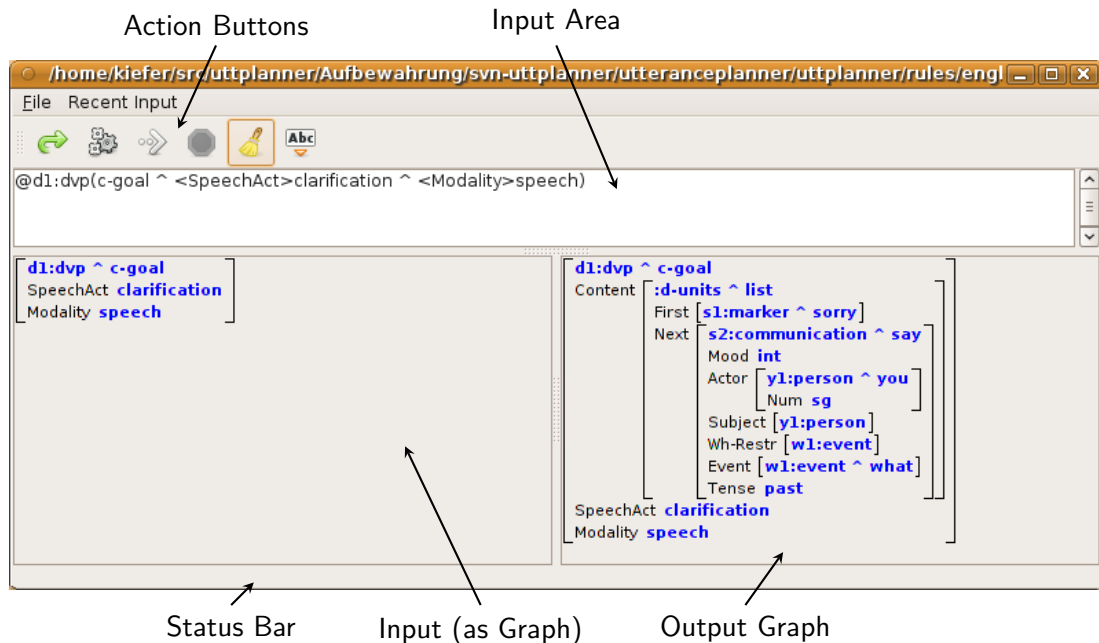


Figure 4: Content Planner Main Window

4.1.1 File Menu

When running in GUI Mode, it is not necessary (put possible) to specify the project file on the command line. The file can also be opened using the **File > Open** menu item. **New** will open a completely fresh planner with empty rule set, which means that a rule file must be opened from the new window for the planner to work. The **File > Recent Files** submenu will contain the last project files that have been opened.

4.1.2 Button Toolbar

The button toolbar is located below the menu bar, and contains buttons for, in this sequence:

- Reloading the rule set (if it has been changed in an editor)
- Processing the logical form in the input area
- Starting processing of the logical form in tracing mode, which opens a new modal trace window that is described below
- Emergency Stop in case the process is in an infinite loop
- Wipe the input area clean
- Running the CCG realizer for the last output LF, provided a CCG grammar was given on startup

If the input contains ill-formed input, and the *Process* or *Start Trace* button has been pressed, the input area gets red and the caret will be located at the point where the error seemingly occurred. In addition, the status line (at the bottom of the window) contains an error message describing the error.

4.2 Trace Window

Figure 5 shows the trace window that will be opened when the trace button in the main window is pressed. It allows to follow the processing of the content planner in single or larger steps, and even going backwards and forwards using the scroll bar at the bottom of the window.

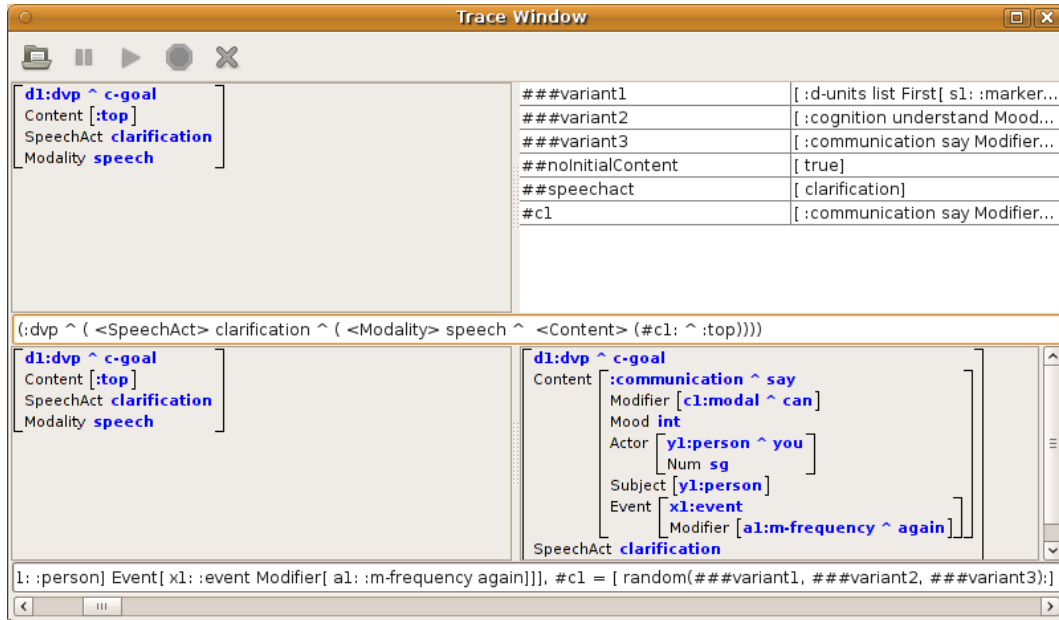


Figure 5: Trace Window

The trace window consists of two main sections, the *match* section, which is in the upper half and consists of the LF window in the upper left, the bindings window in the upper right

and the text window below these. This text area shows the matching part of the rule that is currently ‘processed’ and has been matched successfully against the structure in the window above it.

Since the processor first executes all rule matches pseudo-parallel to the structure and after that applies the actions one by one, this window will change not as often as the windows in the action section (see also section 9).

The second section of the trace window are the two LF windows in the bottom half and the text area for the action part of the rule below them. The lower left LF window shows the current state of the dag, before the action(s) that are shown in the text area are applied, while the lower right LF window shows the new dag after application of the action(s).

If an Emacs connection has been established, double-clicking into the match or action text area will open the appropriate rule file and jump directly to the location of the rule whose application is currently shown.

Additionally, there is a button toolbar at the top of the window, which allows you to perform the following actions:

- Jump to the current rule in the editor (the same as double clicking into the text areas)
- Suspend processing for very long or infinite runs
- Continue suspended processing
- Emergency stop (finishes processing)
- Close the trace window

5 General Description

The new utterance planner module is very much like a specialized graph rewriting system for logical forms as they are used in OpenCCG. A set of rules is specified by the grammar writer which is applied to every part of an input logical form, which is interpreted as a directed graph where the nominals and feature values are the nodes and the features and relations are the (named) edges.

The transfer of nodes is realized using local and global variables, which store information either during the application of a single rule, or during the whole rewrite process, respectively.

6 Rules

Rules have the following form:

$$\underbrace{match^+}_{\text{matching part}} \rightarrow \underbrace{action (, action)^*}_{\text{action part}} .$$

6.1 Matching Part

The reduced BNF for one match looks like this:

<i>match</i>	\rightarrow	<i>pathmatch</i> (\sim) <i>match</i> <i>pathmatch</i>
<i>pathmatch</i>	\rightarrow	< ID > [<i>pathmatch</i>] <i>simplematch</i>
<i>simplematch</i>	\rightarrow	<i>atomic</i> : <i>atomic</i> [: <i>atomic</i>] ! <i>pathmatch</i> (<i>match</i>) = (<i>var</i> <i>gvar</i>) : ((<i>gvar</i> <i>funcall</i>) ~ <i>match</i>)
<i>atomic</i>	\rightarrow	ID <i>var</i>
<i>var</i>	\rightarrow	#ID
<i>gvar</i>	\rightarrow	##ID
<i>funcall</i>	\rightarrow	ID (<i>arg</i> (, <i>arg</i>) [*]) ID ()
<i>arg</i>	\rightarrow	ID STRING # <i>var</i> <i>gvar</i> <i>path</i> <i>funcall</i>
<i>path</i>	\rightarrow	(< ID >) [*]

The match part consists of atomic match expressions which can be combined with conjunction, disjunction and negation operators to obtain complex match expressions:

- **nom**: matches the nominal with name **nom**
- **:type1** matches if the node has type **type1**
- **prop** matches either the proposition value **prop**, or the value under some feature
- **<fname>** matches if **fname** either exists as feature or relation

Additionally, *local variables* also form atomic match expressions. They can bind either whole nodes, type or proposition values, and can also be used several times in a match expression to check that the value of the node is (structurally) equal to what is bound to the variable. Local variables start with a single hash (#), followed by a valid name. The syntax is as follows:

- **#v**: matches a whole node (nominal).

- `:#v` matches the type value of a nominal.
- `#v` matches the proposition value, or the value under a feature.
- `= #var`: checks if the node that is matched is *identical* to what is bound to the variable *var*. An unbound variable will always generate a match failure for this test.

Alternatively, a (bound) global variable can be used for the above checks. Global variables, in contrast to local variables, must begin with two hash characters, e.g., `##global_var`. Global variables can not be bound on the match side, but their values can be checked against the local value as for local variables.

There are two binary and one unary combination operator to combine atomic matches into more complex match expressions:

- `^` for conjunction
- `|` for disjunction
- `!` for negation

Parentheses can be used to group subexpressions.

There are also some short forms of conjunctions following the logical form syntax that do not require the conjunction operator, such as:

`<feat>(nom:#type2)`

matches a node that has a relation `feat` which points to a node with nominal name `nom` and binds the type of that nominal to the local variable `#type2`.

The match conditions, which may be empty, will always be applied to the current node that is under consideration, except if a parenthesized condition with the `~` operator is used. Then, the operand to the left must be either a global variable or a function call, and either the value bound to the variable or the return value of the function call will be matched against the match conditions to the right of the operand. If the global variable is not bound, the match is executed as if it were applied to an independent, empty node.

There is a set of build-in functions, but there is also the possibility of providing your own custom functions via plugins. For a description of the build-in functions and the plug-in interface see sections 7 and 8, respectively. For a detailed description of how the matching and modification of the dag works, see section 9.

6.2 Actions

The BNF for one action looks like this:

<i>action</i>	→	<i>appoint</i> ! < ID > <i>appoint</i> = <i>rexpr</i> <i>appoint</i> ^ <i>rexpr</i>
<i>rexpr</i>	→	<i>rpathepr</i> ^ <i>rexpr</i> <i>rpathepr</i>
<i>rpathepr</i>	→	< ID > <i>rpathepr</i> <i>rsimpleexpr</i>
<i>rsimpleexpr</i>	→	<i>atomic</i> : [<i>atomic</i>] <i>atomic</i> "STRING" (<i>rexpr</i>) ID (<i>args</i>)
<i>atomic</i>	→	ID <i>var</i>
<i>var</i>	→	# ID <i>j</i> ## ID [< ID > (, < ID >)*] ### ID
<i>appoint</i>	→	# <i>var</i>
<i>args</i>	→	ε <i>atomic</i> (, <i>atomic</i>)*

An action consists of the following elements:

- first the specification of the place it is applied to
- an optional feature path
- an operator specifying the kind of action that is to be performed (addition, replacement, deletion)
- the structure that specifies what is added, replaced or deleted.

The place where to apply the action can be either a local or global variable, or a single hash character which represents the node against which the rule is matched.

For addition and replacement, full or partial logical forms, also containing local variables, can be specified. Deletion can currently only be applied to single features or relations.

Minimal examples for the three operations are:

Addition	<code># ^ <feat> val</code> Add a feature / value pair to the current node
Replacement	<code>#var = <feat> #var</code> this would place the nominal that was bound to the local variable <code>var</code> in the matching part under the feature <code>feat</code> instead of its former location.
Deletion	<code># ! <p3></code> delete feature <code>p3</code> from the node that is currently processed

Assignment and replacement of global variables can also be used to inhibit rules, or force a certain ordering on the application of rules (see the section about matching, especially against the content of global variables). In addition, a global variable can be followed by a nonempty path of features, providing a map-like structure. On the match side, these substructures are treated like features of structured nodes and have to be extracted for binding or testing with the usual conjunction and path syntax.

In addition to the local and global variables, *right hand side local variables* can be used in multi-part action specifications. They start with three hash characters and their purpose is three fold:

- provide a means of specifying complex replacements in several steps for clarity
- create temporary structured nodes as arguments to functions.
- establish coreferences in complex nodes on the right hand side, because you are not allowed to use unbound local variables

If variables are used in the replacement of additions and assignments, no matter which kind of variables that is, you have to keep the following in mind:

- if you use `#var:`, the complete node bound to the variable will be added to the specified point of application (which can be an embedded path)
- if you use `:#var` or `#var` or `<feat>#var`, the value bound to the variable must be of a special form, because you use the content of `var` to specify an atomic value:

1. Either it must be a simple (i.e., atomic) value, i.e., a variable bound to a proposition, type, or feature value, *or*
2. It contains a complex node that has a feature with the same name as specified in the replacement, that is, a type, proposition or **feat** feature, respectively

If the above conditions are not met, an error will be signalled.

This does not mean that you can assign a type value only to a type, you can move the atomic values around, if you do it the right way. So the following will always work (you take the type and put it under feature **x**):

```
:#var -> # ^ <x> #var.
```

This will work only if the node bound to **var** has a valid type and proposition:

```
#var: -> # ^ <y> ( :#var ^ #var )
```

6.3 Some Simple Examples

Test for a disjunction of types, and the existence of the **Shape** feature, add a new nominal with the feature's value as proposition and delete the **Shape** feature itself, to avoid infinite recursion. Since the same relation can be added multiple times, the rewriting process would never reach a fix point and, consequently, not stop.

```
(:entity | :physical | :e-substance) ^ <Shape> #v
->
# ^ <Modifier> (shape:q-shape ^ #v),
# ! <Shape>.
```

A similar example, but now, the whole node under **TopIn** is moved under the **Anchor** feature, which specifies with the parentheses and the nominal specification with the colon after the variable that it is a relation. Note that under **Modifier**, no specification of the nominal name is necessary, which results in a new, unique nominal name for that node.

```
(:entity | :physical | :e-substance) ^ <TopIn> ( #topin: )
->
# ^ <Modifier> (:m-location ^ in ^ <Anchor> ( #topin: )),
# ! <TopIn>.
```

In the next example, a local variable is used only in the action part multiple times to indicate the coreferencing of two nominals. The whole content under **Content** will be replaced due to the replacement specification for the matched local variable **c1**

```
:dvp ^ <Relation> accept ^ <Content> ( #c1:marker )
->
#c1 = (:ascription ^ <Actor>(#i1:person ^ I)
      ^ <Patient>(:entity ^ what)
      ^ <Subject>( #i1: )).
```

The next example adds the result of a function call to the current node. The function gets as argument the structure that is bound to the variable `c1`. A single `#` is also a valid argument to a function and represents the node that is currently processed, like in the initial place of the action specification.

```
:dvp ^ <Relation> accept ^ <Content> ( #c1:marker )
->
# ^ <Content> modify_marker(#c1).
```

Functions can also be used on the matching side (see section 6.1), as in the next example:

```
:dvp, random(one, two, three) ^ two
->
# = <Content> ( <Answer> positive ).
```

Also look into the file `presentation-examples.trf`, which contains much more elaborate examples with explanations attached to it.

7 Build-in Functions

7.1 Mathematical Functions

Mathematical functions expecting `double` arguments will return `NaN`, if one of the arguments is not a valid number.

- `add`, `sub`, `div`, `mult` expecting two `double` arguments, returning a `double` argument, the usual binary operators
- `lt`, `gt`, `lteq`, `gteq`, `eq` expecting two `double` arguments, returning either one or zero, representing true and false, for comparison of numbers
- `not` returns zero if the (`double`) argument is non-zero, one otherwise
- `neg` is unary minus.

7.2 Other Functions

- `concatenate` concatenates an arbitrary number of strings and returns the result
- `clone` clones the argument node, to get independent copies of parts of the current structure
- `random` takes an arbitrary number of arguments and randomly returns one of them
- `throwException` takes an (optional) message argument and throws a `PlanningException` with the given message. This will end the processing of the current input.
- `warning` logs a warning message on to the `UtterancePlanner` main logger, and always returns `true`

8 Adding New Functions

New functions can be added using a lightweight plug-in mechanism. All functions must be subclasses of `Function`, if they work on the internal structures of the planner directly, or of `LFFunction`, if they work with `LogicalForm`, as provided with the separate `CPlanWrapper` class. The methods that must be overwritten for both classes are:

```
public abstract String name();  
public abstract int arity();
```

`name` must return the name as it is used in the rules, and `arity` the expected length of the argument list.

Additionally, the abstract method `Object apply(List args)` must be implemented for every subclass of `Function`, except for `LFFunction`, where `LogicalForm applyLfFunction(List args)` must be implemented instead. The length of `args` matches the arity specified by the function. If the functions are not directly compiled in and registered at the `FunctionFactory`, they can be provided as plug-ins. The class files of the functions have to be added to a jar-archive such that the path where they are stored in the archive matches the package prefix.

```
$ jar tf plugin-jars/cplan-plugins.jar
```

```
META-INF/  
META-INF/MANIFEST.MF  
de/dfki/lt/tr/dialogue/cplanwrapper/ChangePropFunction.class
```

Figure 6: `de.dfki.lt.tr.dialogue.cplanwrapper.ChangePropFunction` properly put into a `.jar` file to use as plug-in function

These jars have to be put into one directory, which is passed to the constructor of the `UtterancePlanner` class as second argument. The constructor will then load all appropriate classes (subclasses of `Function`) on startup.

9 Processing Regime

Processing is divided into different stages. Every stage has its own set of rules, which are loaded from the rule files specified in different sections of the project file (see 3 for further information), and essentially behaves in the same way.

For every stage, the current processing engine proceeds as follows:

- Traverse the structure in preorder sequence, starting at the root node
- At every node, match all rules against the node in the order in which they have been loaded

If the match was successful:

store the rule together with the local bindings of the match for later execution of the rule's actions

- Execute the actions for the stored rule / binding pairs in the order in which they were stored, which is the order in which the matches occurred.

Since the matches are applied before any changes are made, any applicable match will be found. When the changes are applied afterwards, it is possible that some of them will not have any effect on the input structure, say, if something is added to a node that has previously been deleted.

This process is iterated until a fixpoint is reached, i.e., the structure does not change by applying the rules as described. If the current stage was not the last stage, the structure is then processed with the next set of rules, until all stages have been applied. Every stage is only applied once, there is no fixpoint loop around the stages. The stages are applied in the order in which they appear in the project file, independent of their names.