The University of Sheffield
Department of Computer Science

Adaptive Information Extraction from Text

Amilcare V2.4

USER MANUAL

Fabio Ciravegna

Department of Computer Science, University of Sheffield

Regent Court, 211 Portobello Street

Sheffield S1 4DP UNITED KINGDOM

http://www.dcs.shef.ac.uk/~fabio/

F.Ciravegna@dcs.shef.ac.uk

1 AMILCARE USER MANUAL

1.1.1 Introduction

Amilcare is an adaptive IE system, i.e. it uses machine learning to adapt to new applications/ domains. It is rule based, i.e. its learning algorithm induces rules that extract information. Rules are learnt by generalizing over a set of examples found in a training corpus annotated with XML tags. The system learns how to reproduce such annotation via Information Extraction.

Amilcare is based on an adaptive methodology that meets most of the requirements mentioned in the previous section, in particular: (1) portability by a wide range of users, from naive users to IE experts; (2) ability to cope with different types of texts (including mixed ones); (3) possibility to be inserted in the usual user annotation environment providing minimum disruption to usual annotation activities; (4) portability with reduced number of texts.

1.1.2 Modes of Operation

Amilcare can work in three modes: training mode, test mode and production mode.
The **training mode** is used to induce rules, so to learn how to perform IE in a specific application scenario. Input in training mode is: (1) a scenario (e.g. an ontology in the SW); (2) a training corpus annotated with the information to be extracted. Output of the training phase is a set of rules able to reproduce annotation on texts of the same type.

The **testing mode** is used to test the induced rules on an unseen tagged corpus, so to understand how well it performs for a specific application. When running in test mode Amilcare first of all removes all the annotations from the corpus, then re-annotates the corpus using the induced rules. Finally the results are automatically compared with the original annotations and the results are presented to the user. Output of the test phase is: (1) the corpus reannotated by the system; (2) a set of accuracy statistics on the test corpus: recall, precision and details on the mistakes the system does. Amilcare uses an internal scorer, but it is also compatible with some standard scorers such as the MUC scorer [Douthat 1998]. During testing it is possible to decide to retrain the learner with different system parameters in order to tune its accuracy (e.g. to obtain more recall and/or more precision). Tuning takes a fraction of time with respect to training.

The **production mode** is used when an application is released. Amilcare annotates the provided documents. If a user is available to revise its results, the learner uses the user corrections to retrain. The training/test/production modes can actually be interleaved so to produce an active learning based annotation. In active learning [Califf and Mooney 1997] user annotation and system annotation are interleaved in order to

minimize the amount of user annotation. As we will see in the following this greatly reduces the burden of document annotation.

Amilcare's default architecture includes the connection with Annie, Gate's shallow IE system [11] which performs tokenization, part of speech tagging, gazetteer lookup and named entity recognition. Any other preprocessor can be connected via the API. The preprocessor is also the only language-dependent module, the rest of the system being language independent (experiments where performed in English and Italian).

1.1.3 The learning algorithm

Amilcare is based on the $(LP)^2$ algorithm [Ciravegna 2001a], a supervised algorithm that falls into a class of Wrapper Induction Systems (WIS) using LazyNLP [Ciravegna 2001b]. Unlike other WIS [Kushmerick et. al. 1997; Muslea et. al. 1998; Freitag and Kushmrrick 2000; Freitag and McCallum 1999], LazyNLP WIS use linguistic information. They try to learn the best (most reliable) level of language analysis useful (or effective) for a specific IE task by mixing deep linguistic and shallow strategies.

The learner starts inducing rules that make no use of linguistic information, like in classic wrapper-like systems. Then it progressively adds linguistic information to its rules, stopping when the use of linguistic information becomes unreliable or ineffective. Linguistic information is provided by generic NLP modules and resources defined once for all and not to be modified by users to specific application needs. Pragmatically, the measure of reliability here is not linguistic correctness (immeasurable by incompetent users), but effectiveness in extracting information using linguistic information as opposed to using shallower approaches. Unlike previous approaches where different algorithm versions with different linguistic competence are tested in parallel and the most effective is chosen [Soderland 1999], lazy NLP-based learners learn which is the best strategy for each information/context separately. For example they may decide that using parsing is the best strategy for recognizing the speaker in a specific application on seminar announcements, but not the best strategy to spot the seminar location or starting time. This has shown to be very effective for analyzing documents with mixed genres, e.g. web pages containing both structured and unstructured material, quite a common situation in web documents [Ciravegna and Lavelli 2001].

$(LP)^2$ induces two types of symbolic rules in two steps: (1) rules that insert annotations in the texts; (2) rules that correct mistakes and imprecision in the annotations provided by (1). Rules are learnt by generalizing over a set of examples marked via XML tags in a training corpus.

A **tagging rule** is composed of a left hand side, containing a pattern of conditions on a connected sequence of words, and a right hand side that is an action inserting an XML tag in the texts. Each rule inserts a single tag, e.g. < /speaker>. As positive examples the rule induction algorithm uses XML annotations in a training corpus. The rest of the corpus is considered a pool of negative examples. For each positive example the algorithm: (1) builds an initial rule, (2) generalizes the rule and (3) keeps the k best generalizations of the initial rule. In particular $(LP)^2$'s main loop starts by selecting a tag in the training corpus and extracts from the text a window of *w* words to the left and *w* words to the right. Each piece of information stored in the *2*w* word window is transformed into a condition in the initial rule pattern, e.g. if the third word

is "seminar", a condition word3="seminar" is created. Each initial rule is then generalized (see sections 1.1.4) and the k best generalizations are kept: retained rules become part of the best rules pool. When a rule enters such pool, all the instances covered by the rule are removed from the positive examples pool, i.e. they will no longer be used for rule induction ((LP)^2 is a sequential covering algorithm). Rule induction continues by selecting new instances and learning rules until the pool of positive examples is empty. Some tagging rules (contextual rules) use tags inserted by other rules. For example some rules will be used to close annotations, i.e. they will use the presence of a < speaker> to insert a missing < /speaker>. In conclusion the tagging rule set is composed of both the best rule pool and the contextual rules.

Tagging rules when applied on the corpus may report some imprecision in slot filler boundary detection. A typical mistake is for example "at < time> 4 < /time> pm", where "pm" should have been part of the time expression. (LP)^2 induces rules for shifting wrongly positioned tags to the correct position. It learns from the mistakes made in tagging the training corpus. **Correction rules** are identical to tagging rules, but (1) their patterns match also the tags inserted by the tagging rules and (2) their actions shift misplaced tags rather than adding new ones. The induction algorithm used for the best tagging rules is also used for shift rules: initial instance identification, generalization, test and selection. Shift rules are accepted only if they report an acceptable error rate. In the testing phase, information is extracted from the test corpus in four steps: initial tagging, contextual tagging, correction and validation. The best rule pool is initially used to tag the texts. Then contextual rules are applied in a loop until no new tags are inserted, i.e. some contextual rules can match also tags inserted by other contextual rules. Then correction rules correct imprecision. Finally each tag inserted by the algorithm is validated. In many cases there is no meaning in producing a start tag (e.g. < speaker >) without its corresponding closing tag (< /speaker >) and vice versa, therefore uncoupled tags are removed.

1.1.4 Rule Generalization

The initial rule pattern matches conditions on word strings as found in a window around each instance. This type of rules is the typical type of rules wrapper induction systems produce. We have previously shown [Ciravegna 2001b] that these types of rules are suitable for highly structured texts, but quite ineffective on free texts, because of data sparseness implied by the high flexibility of natural language forms. It is therefore important to generalize over the plain word surface of the training example. The initial rule pattern is integrated with additional conditions on the results of the linguistic preprocessor and the ontology associated with the texts (if given). This means that conditions are set not only on the strings (word="companies"), but also on its lemma ("company"), its lexical category ("noun"), case information ("lowercase"), a list of user defined classes from a user-defined dictionary or a gazetteer, annotations by a named entity recognizer and subsumption hierarchy as given in the ontology. Figure 2 summarizes the new form of the initial rule.

Condition	Additional Knowledge				Action
Word	Lemma	LexCat	Case	SemCat	Tag
the	the	det	low		
seminar	seminar	noun	low		
at	at	prep	low		
4		digit	low		<time>
pm		noun	low	timeid	
will	will	verb	low		

Figure 1 – An example of an initial rule

The initial rule effectiveness is compared against a set of more general rules obtained through a general to specific beam search that starts modeling the annotation instance with the most general rule (the empty rule matching everything in the corpus) and specializes it by greedily adding constraints. Constraints are added by incrementing the length of the rule pattern, i.e. by adding conditions on terms (matching strings or part of the knowledge provided by the linguistic preprocessor). The specialization process stops when the generated rules have reached a satisfying accuracy and coverage and any further specialization does not improve precision. In this way the rules that incorporate different levels of linguistic knowledge (from deep linguistic to word matching) are put in direct competition and just the best combination of *k* rules survive. Figure 2 shows a possible generalization of the rule in Figure 1.

Condition					Action
Word	Lemma	LexCat	Case	SemCat	Tag
	at				
		digit			<time>
				timeid	

Figure 2 – Initial rule generalized

The choice of the level of linguistic knowledge is local to the specific instance we are trying to model (local constraints, e.g. the information is located in a table), but it also depends on the ability to generalize to cover other instances (global constraints, e.g. it is a noun followed by a capitalized word). For this reason the approach satisfies the requirement of coping with different types of texts including mixed types, because it is able to model different parts of texts in which different information is located using different (linguistic) strategies. Details of the algorithm can be found in [Ciravegna 2001b].

1.1.5 Installing Amilcare

Requirements for installation:

1. Windows, Unix or Linux OS: at least 256M RAM in operational conditions (128 for demo)
2. Java version 1.4 (there is a special version for 1.3.1)

Amilcare comes with a Wizard for installation called:

1. AmilcareWizard.bat for Windows
2. AmilcareWizard.sh for Unix or Linux

A number of files are provided for installation, mainly jar files, as Amilcare is written in Java.
Be sure to have java installed and running on your machine (***check your version against the minimum requirements above***) and its path set correctly (refer to the java instruction if unsure, see http://www.java.sun.com).

Install all the content of the archive you have been provided (both Wizard and Amilcare) in the directory where you want to install the tool. Then run the wizard. A window will appear:

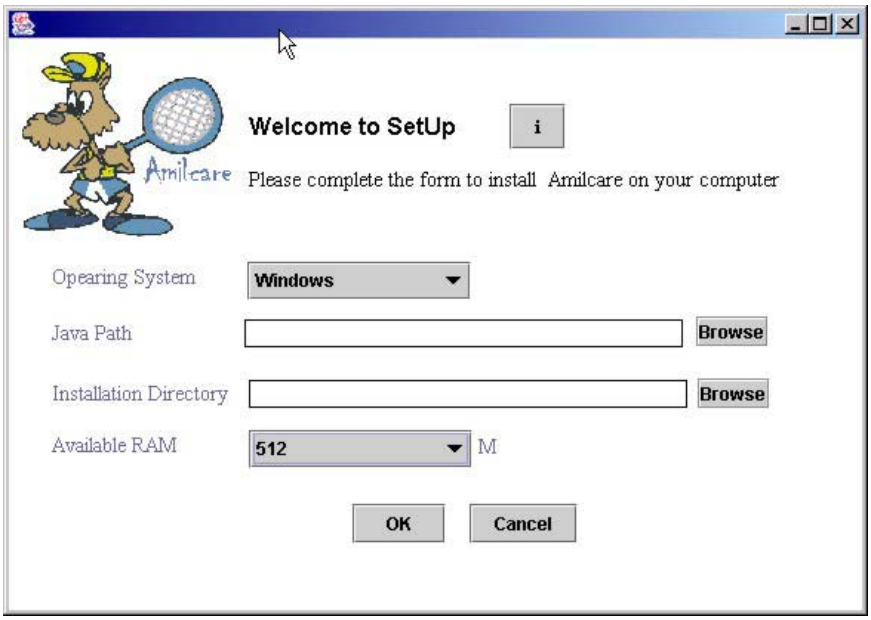

Figure 3 - Wizard Set Up window

Select the correct OS and choose an installation directory (check read/write permissions!!!). Insert the java path (this is the main java directory; the wizard will look for the lib/ directory). Insert the amount of memory your computer has (or you want to allocate to Amilcare).
Then select OK.
The installation is done.

1.1.6 Running Amilcare

The installation will create in the selected directory a file called:

1. Amilcare.bat on Windows
2. Amilcare.sh on Unix/Linux

In order to run Amilcare, just execute that file. A window will appear for a few seconds during initialisation, then it will disappear and the following will appear.

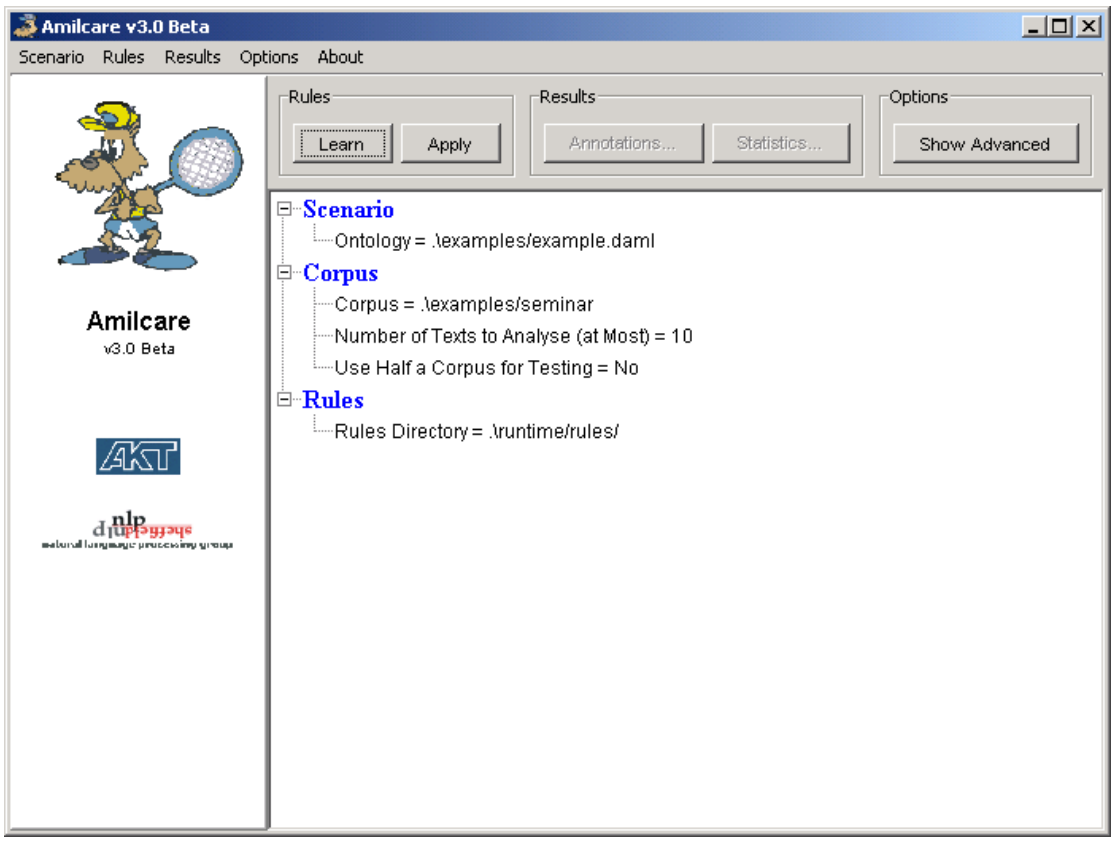

Figure 4 – Amilcare in Normal Mode

This is Amilcare setup interface to be used to define a number of parameters and to run it. There are two **set up modes**: normal and advanced. The "**Show Advanced**" button allows toggling between the modes.

Independently from the mode, the menus are invariant. They allow to:

1. Scenario: save and load the running parameters. In this way it is possible to save a configuration and reload it when necessary;
2. Rules: run amilcare (i.e. learn or extract). Selecting this menu item is equivalent to press the buttons to the right hand side of the main window. An additional item that is selectable only after a run, allows inspecting the rules and their coverage as explained below;
3. Results: view the results of execution of Amilcare. Selecting this menu item is equivalent to press the buttons in the middle of the main window;
4. Options: toggle between the two interaction modes (normal and advanced). It also allows activating the Gate interfaces in case of needed debugging;
5. About:: provides information about Amilcare and Gate.

The parameters that are set by default when you run Amilcare first time point to default locations of files and resources that come with the distribution, including example ontologies, example corpus, etc.

1.1.7 Set Up in Normal Mode

In normal mode the number of parameters to be set are limited. This is the mode in which a non-expert user will operate.

There are three elements to be defined: Scenario, Corpus and Rules. Each one is defined by a number of features, clickable for value insertion:

1. **Scenario**: a scenario defines the information to be extracted, or – better – the tags that the user will insert in the training corpus. It allows declaring the list of XML tags the user will use for tagging the corpus. A tag is a string [a-zA-Z0-9]$^{+}$ enclosed in angular brackets (e.g. "<speaker>" is a tag). If the user specifies an ontology, all the tags in the ontology are considered. Otherwise, the user can edit the tags individually. Clicking the feature value will pop up the following window, which allows adding, removing and importing tags from ontologies:

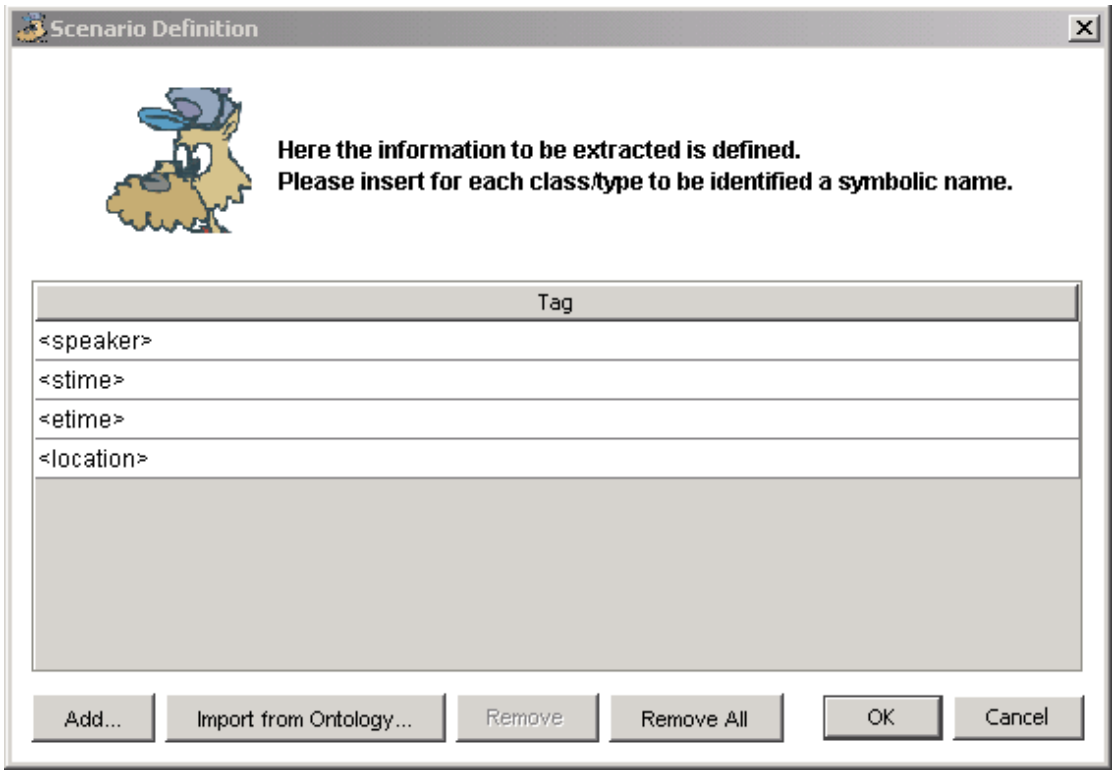

Figure 5 – Scenario Definition in Amilcare

2. **Corpus**: The set of texts from which Amilcare learn how to extract information or is actually run to extract information. The actual action depends on the button that is clicked when the system runs after set up. The corpus can be either a directory (in which case all the files in the directory will be selected) or a single file. Files can be of any type accepted by Gate (e.g. XML. SGML). Also txt files are acceptable, even if they include partial xml markup (e.g. if they just contain the tags for the information to be extracted but not the regular <xml> headers). Amilcare will try to convert them to standard XML **but it could fail**, because it has no standard converter. An annotated corpus

can be freely used for testing, as Amilcare – when used in test mode – strips the annotations before running. The annotations will be then used to compare the system results with the user-defined results as we will see below;
It is possible to decide to use **half of the corpus** for training and half for testing. In this case the texts used for training (Learn button) will always be different from those used for testing (Apply button). The strategy implemented is: texts are numbered and even texts are used for training. Do not rely on the assumption that this is the strategy as it is not guarantee to hold in the future.

3. **Rules**: the learning algorithm produces rules (that can be modified by the user as we will see below). Such rules are stored in a number of files. Here you have to select the directory where the rule files will be stored.

1.1.8 Set Up in Advanced Mode

This mode allows the user to deeply control the learner. The user is warned to be very careful in changing parameters whose role is not completely understood, as the system could stop working in a reasonable way. See figure 3 for details.

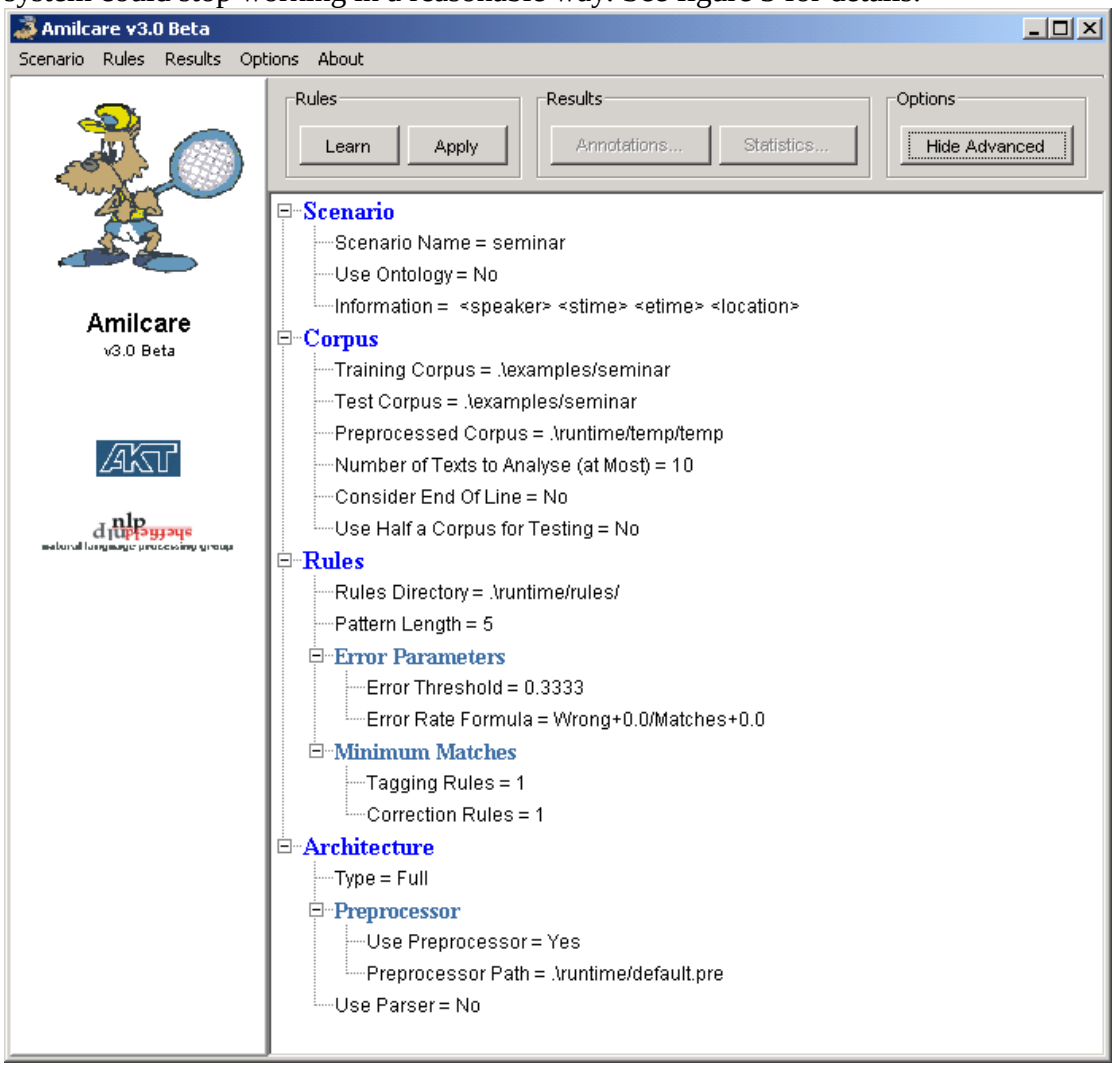

Figure 6 - Advanced mode

1. **Scenario**: same as in normal mode
2. **Corpus**: it is possible to define different corpora for training and test. Advanced features:

a. End of Line: Amilcare can either consider that the end of line is relevant or simply consider it as a blank space. Typically Eol are to be used only in txt texts that present manual formatting (e.g. email announcing conferences typically are txt files where the name of the location is centred, followed by the dates in the following line, etc.)
b. Pre-processed corpus: Amilcare takes the corpora passed for input to Gate (www.gate.ac.uk) which performs a number of pre-processing steps, including tokenization, sentence splitting, part of speech tagging and use of a gazetteer. Gate writes its results on a file that Amilcare loads before running. Here it is possible to select the name of the file. Note that it is this file that is used when the pre-processor is disabled (see below).

3. **Rules**: additional features**:**
   a. **Pattern Length**. It is possible to influence the maximum length of the induced rules' pattern. The **maximum** pattern length is given by the number given here multiplied by 2. A longer pattern means more precision, but a longer training time. Suggestion: always use a number between 3 and 5;
   b. **Error Threshold**: Amilcare selects rules using an error threshold given by:

$$\frac{\text{Number of mistakes} + \text{epsylon1}}{\text{Number of matches} + \text{epsylon2}}$$

   The result of this formula is matched against the **maximum error threshold**. If a rule reports a bigger error rate, the rule is discarded;
   c. Error rate formula: allows to set epsylon1 and epsylon2 in the formula above. As a suggestion always leave both to 0.0 when using a limited corpus. Use very small amounts on bigger corpora (e.g. 0.01). Note that epsylon2 **must always be greater than or equal to** epsylon1;
   d. **Minimum Matches:** select rules when they present at least a minimum number of matches. Always use 1 for small corpora. As a general rule never put this above 3 on medium size corpora, but it depends on the task.

4. **Architecture:** it is possible to influence the architecture used:
   a. **Type:** in *lite* mode Amilcare just uses the best annotation rules, in *full*mode it also uses contextual and correction rules. There is generally a considerable difference in using them: the lite version is faster but more imprecise. As a suggestion, use the lite version for testing and move to the full version when you are playing seriously;
   b. **Preprocessor**: It is possible to run just once the pre-processor on a specific corpus and then it can be freely disabled. Amilcare will read the preprocessed file the user selects (see above) and avoid calling Gate for the rest of the time. The preprocessor path expects a Gate application defined with processing resources in a sequence.
   c. Parser: Amilcare does not provide a parser, but it is possible to integrate the results of a chunker (by modifying the preprocessed file). It is possible to enable this feature **but keep in mind that this is just experimental and it is not guaranteed to work**.

1.1.9 Training the System

During training Amilcare develops rules to reproduce the annotations found in the training corpus. To train it, you need:

1. a corpus
2. setup the learner with the proper parameters (see above). When setup is ready, just press the learn rules button

1.1.10 Annotating a corpus

A number of annotation tools can be used for annotation. Melita [Ciravegna et. al. 2002] is an ontology-enabled annotation tool that uses Amilcare. Gate provides its own annotation tool. Also Mitre's Alembic is a good one. Amilcare is also integrated with MnM [Vargas-Vera et. al. 2002] and S-CREAM [Handschuh et. al. 2002]. Please refer to these tools web pages to download them.

Typically a training file is an XML file with annotations inserted. Only the annotations selected defined in the scenario definition (see above) are used to train. The others are used as contextual information in the text (e.g. are considered as words). An example of an XML text is the following:

```
<XML version="1.0" encoding="UTF-8">
Title: seminar announcement:

Speaker:   <speaker>Steven Skiena</speaker>
            Department of Computer Science
            State University of New York, Stony Brook

 Date:   Tuesday, January 14
 Time:      <stime>1:30 pm</stime>
 Place:     <location>Wean Hall 4634</location>

 Topic:     Algorithms for Square Roots of Graphs

The kth power of a graph  G = (V, E), written G^k,
is defined to be the graph having V as its vertex set
with
vertices u, v adjacent in G^k if and only if
there exists a path of length at most k between them.
Similarly, graph H has a kth root G if G^k = H.
Powers of graphs have several interesting properties.
For example, the square of any biconnected graph and the
cube of any connected graph is hamiltonian.

In this talk, we present algorithms to find a square
root G of a graph G^2 for some special classes of graphs.
This is joint work with Yaw-Ling Lin.

 Host: Dana Scott

Appointments can be made through Lydia DeFilippo, x3063,
lydia@cs.cmu.edu.

<speaker>Steve Skiena</speaker> will be at CMU Monday,
January 13, and Tuesday, February 14.
</XML>
```

Note that the following tags are defined for IE: *speaker, stime, etime and location*.

1.1.11 End of training

At the end of training Amilcare will test the developed rules on the training corpus. The accuracy should be quite high (around 100%). If it is not, there are some problems to investigate (e..g. wrong parameter settings).

1.1.12 Extracting Information

Once Amilcare has been trained, it is possible to use the induced rules to extract information from texts. The suggestion is to create a test corpus tagged in the same way as it was the training corpus and run Amilcare on it in test mode. It is safe to do it because Amilcare strips off all the annotations during test, but it uses them to provide accuracy measures to the user.

In order to run the system, just set up the parameters and click the button "Apply". At the end of the run, the button "Annotations" will be enabled. Click it and the window in Figure 7 will appear:

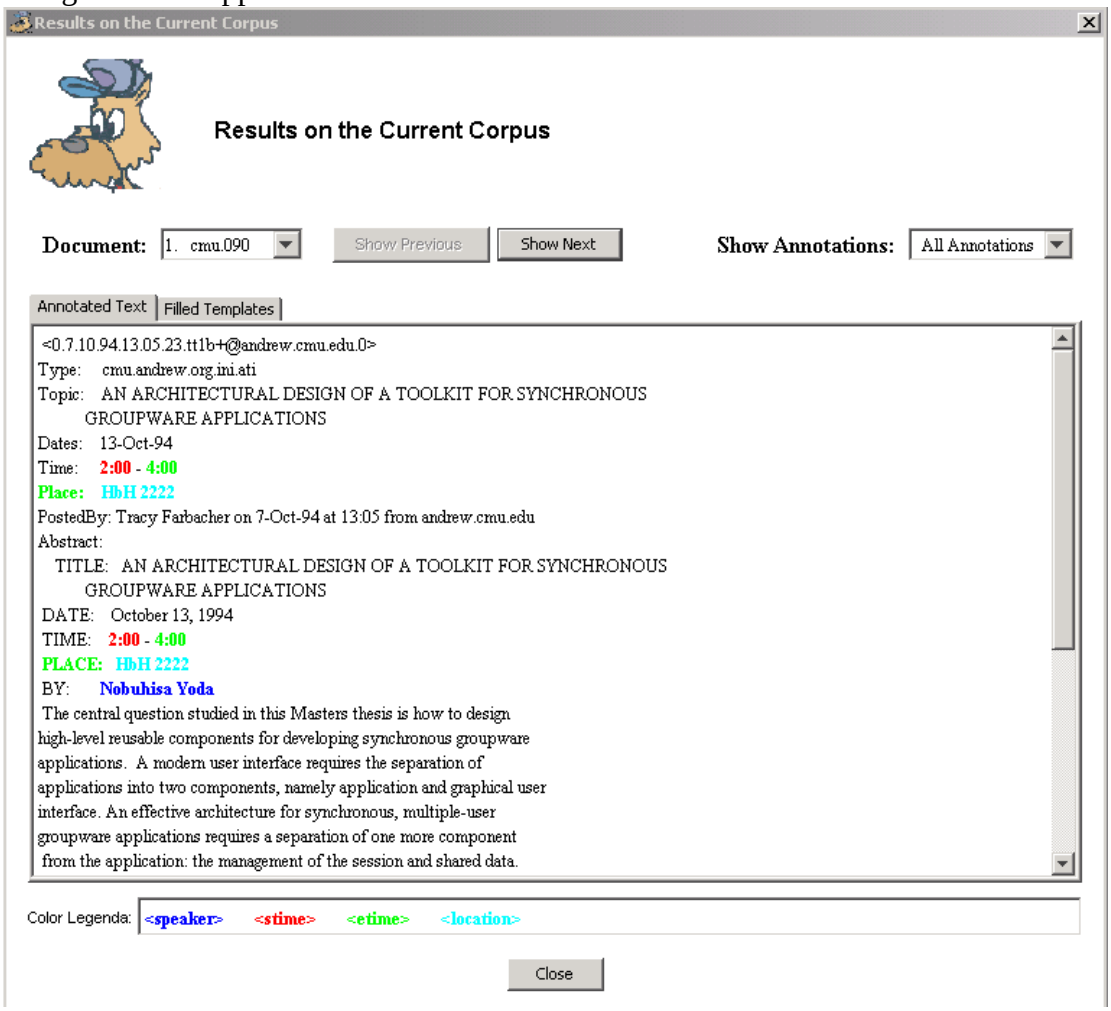

Figure 7 - Result output window

The window presents the following parts:

1. Top-Left: a combo box for selecting the text in the corpus for which the results are displayed (there are corresponding previous/next buttons to navigate the list);
2. It is possible to inspect either the annotated text or the generated file: note that the ability to generate templates is very reduced: Amilcare always supposes there is just one event and fills the template using information found in the page.
3. Below: the text with the extracted information highlighted in different colours (the colour legenda is shown at the bottom).

It is also possible to see the statistics for the current session (button “Statistics”): Amilcare makes available the information about the correctness in retrieving information (e.g. in recognising “speaker”), single tags (<speaker> and </speaker> are treated differently) and templates. Possible are the number of user-defined annotations, actual the number of annotations inserted by Amilcare, Correct/Wrong and Missing are the number s of annotations Amilcare inserts correctly/incorrectly or misses. Partials are solutions partially overlapping the correct solution (e.g. “Mark” is a partial annotation of “John Mark Smith” and viceversa).

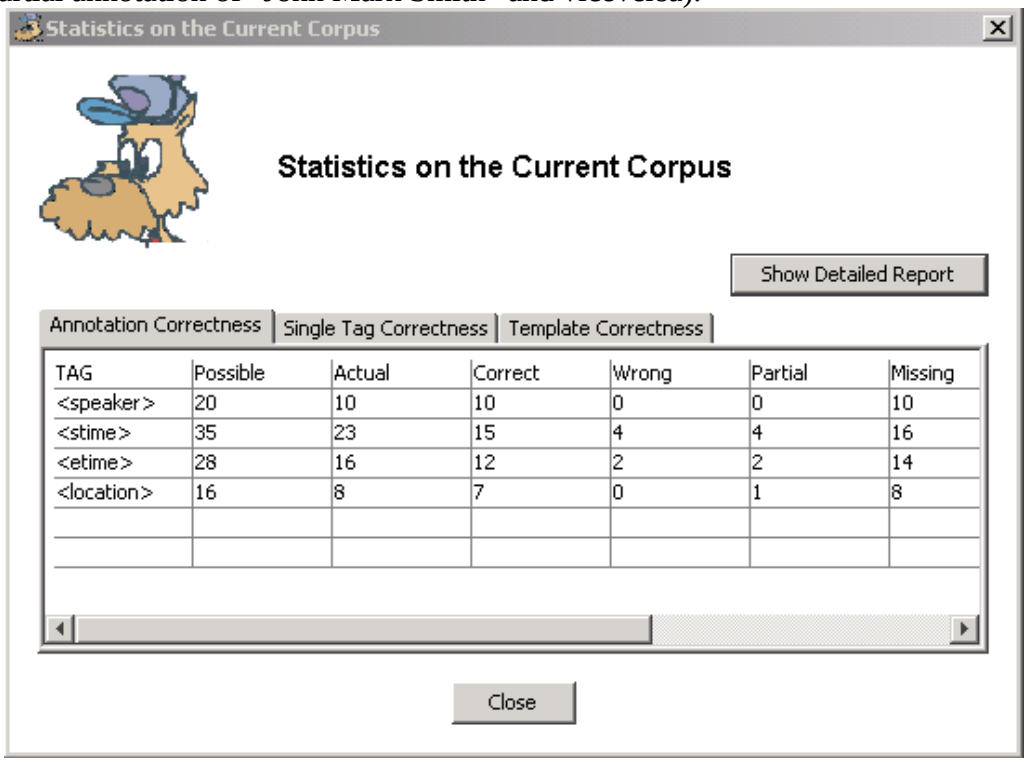

Figure 8 – Statistics for the current session

You are likely to be only interested in the lines in which your tags appear (e.g. speaker, stime, location and etime in the example in figure) and in the columns Recall and Precision.

Please note that the template Correctness TABLE REPORTS A BUGGED TABLE!!!!!

1.1.13 Managing the Induced Rules

It is possible to manage the induced rules by selecting the Edit in the Rules menu. The following window (rule manager) is shown:

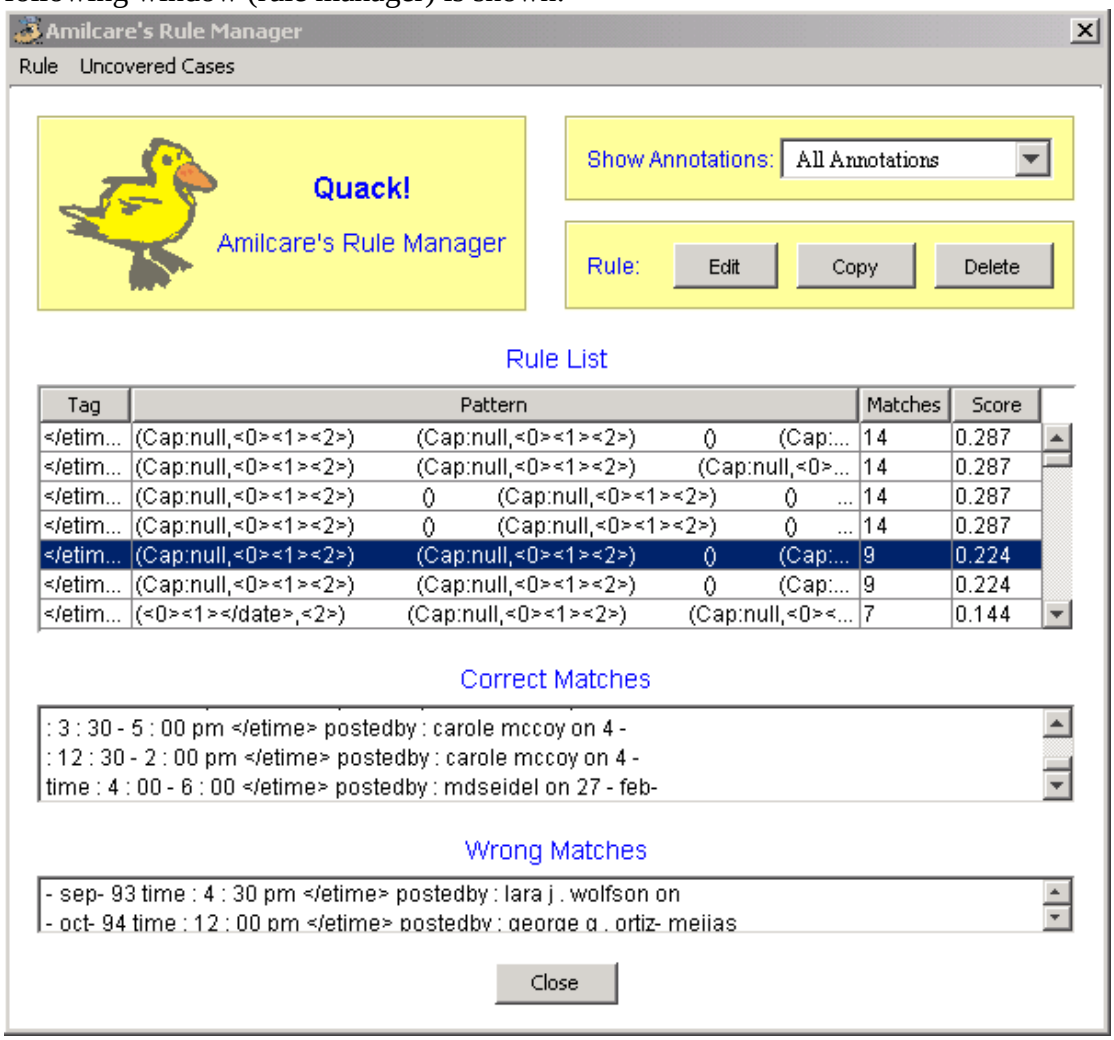

Figure 9 – Amilcare's Rule Manager

In the top panel the list of rules are shown (feel free to adjust the column width in the table). Rules are sorted by tag and decreasing number of matches. For each rule the following information is shown:

1. Tag
2. Pattern: a pattern is a set of conditions on a sequence of words. Each word is represented by parentheses. A word in the text presents a number of features potentially tested by the rules:
    a. Tok (original token cast in lowercase)
    b. Lemma (if a lemmatizer is provided – not provided in Version 1.0 beta)
    c. Pos: part of speech (e.g. noun). The brill's tagset is used.
    d. Sem: a tag as provided by the gazetteer or a user defined dictionary
    e. Cap: capitalization (e.g. uppercase)
    f. Nerc: a tag returned by Annie's named entity recogniser
    g. Oth: typically html tags found in the text.
3. Matches: the number of times a rule was fired on the current corpus;

4. Score: the score (tipically number of errors/number of fires) that the rule has if the corpus is tagged with the correct solutions (otherwise it is 1.0)

In the middle and low panels, the details of the matches are shown, divided with wrong and correct. If the corpus was not tagged all the matches are shown as incorrect. A window in the texts is shown with the annotation inserted by the rule.

It is possible to modify the induced rules either by editing, copying or deleting. Select the rule you want to work on and select the corresponding button (edit, copy or delete). Double-clicking on a rule opens the editor as well. The editor appears as follows:

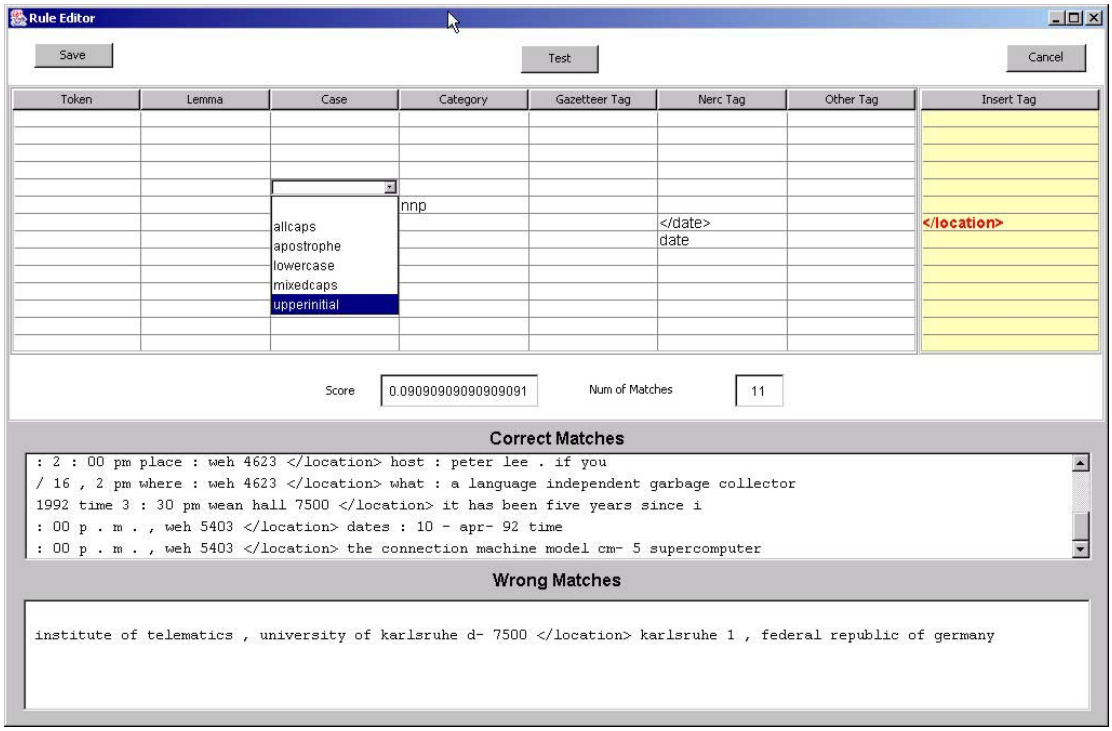

Figure 10 – Rule Editor

Some of the fields can be filled by using some predefined menus. The menu is generated using the values found in the corpus. It is not possible to add values, so edit only after loading a reasonable corpus (some dozens of texts).

It is possible to test the rule before saving it and to see the cases matched and compute the score. When a rule is saved, it is immediately written on the rule file.