|  |
| --- |
| TERNIP: Temporal Expression Recognition and Normalisation in Python |
| This report is submitted in partial fulfilment of the requirement for the degree of MSc in Computer Science with Speech & Language Processing  1st September 2010 |
| Christopher Northwood  *Supervisor: Mark Hepple*  Department of Computer Science, University of Sheffield |

All sentences or passages quoted in this dissertation from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this dissertation have been used with the explicit permission of the originator WHERE POSSIBLE and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this dissertation and the degree examination as a whole.

Name: Christopher Northwood

Signature:

Date: 1st September 2010

Abstract

Abstract goes here

Table Of Contents

[1 Introduction 1](#_Toc270439780)

[2 Background 2](#_Toc270439781)

[2.1 Temporal Expressions 2](#_Toc270439782)

[2.2 Annotation Standards 3](#_Toc270439783)

[2.3 Evaluating Tagger Performance 6](#_Toc270439784)

[2.4 Corpora 6](#_Toc270439785)

[2.5 Temporal Expression Taggers 7](#_Toc270439786)

[3 Problem Analysis 13](#_Toc270439787)

[4 System 14](#_Toc270439788)

[4.1 Architecture 14](#_Toc270439789)

[4.2 Recognition By Rule 16](#_Toc270439790)

[4.3 Normalisation By Rule 22](#_Toc270439791)

[4.4 Document Formats 29](#_Toc270439792)

[5 Evaluation 32](#_Toc270439793)

[5.1 Software Verification 32](#_Toc270439794)

[5.2 Performance Evaluation 32](#_Toc270439795)

[6 Discussion 33](#_Toc270439796)

[7 Bibliography 34](#_Toc270439797)

# Introduction

Introduction to this dissertation

# Background

## Temporal Expressions

Temporal expressions, or “timexes”, are “phrases or words that refer to times, where times may be points or durations, or sets of points or durations” (Ahn, Rantwijk, & de Rijke, 2007), and the identification and interpretation of these timexes is an active topic of research. Temporal expressions are a very rich form of natural language, with Pustejovsky, et al. (2003) identifying three main types of temporal expressions:

* “Fully-specified temporal expressions (e.g., June 11, 1989, or Summer 2002);
* Underspecified temporal expressions (e.g., Monday, next month, two years ago);
* Durations (e.g., three months, two years).”

Most systems identify two distinct, but related, tasks for the identification of timexes. The first is that of recognition, which simply identifies which phrases in some text are temporal, that is, refer to some point in time. The second task is that of normalisation, which takes the identified expressions, and attempts to resolve them into some standard format (e.g., ISO 8601) to anchor the expression at a particular point in time (Ahn, Adafre, & de Rijke, 2005).

Interest in recognition of temporal expressions grew out of the field of information extraction. The Message Understanding Conferences of the 1990s dealt with the task of named entity recognition, and early timex recognition systems simply treated timex recognition as a part of named entity recognition (Krupka & Hausman, 1998). Temporal expression recognition is clearly an important task for information extraction; however, identification of temporal expressions by itself is of limited usefulness.

Normalisation is important to allow for further processing, such as construction of event chronologies, or in question answering systems, and is an important part of natural language understanding. In the phrase “do you want to go to the pub at 7?”, a human may recognise the expression “7” as a temporal expression and normalise that to a particular point in time based on context of the current date, and the background knowledge that visits to public houses are more likely in the evening.

Mani & Wilson (2000) introduced a prominent system that used a rule-based system for recognition and normalisation using the technique of establishing tense. Following this, the Time Expression Recognition and Normalization (TERN) evaluation as part of the 2004 Automated Content Extraction (ACE) programme (MITRE, 2004) was the first competition that dealt specifically with recognition and normalisation as a distinct task from named entity recognition.

Following this early work and the TERN competition, interest in temporal expressions has grown, with multiple systems built and many approaches to recognition and normalisation investigated. These systems and approaches are discussed further in section 2.5.

Simple normalisation of temporal expressions is not enough to capture the full range of temporal information available in a body of text, as a considerable amount of temporal information is implicit (Verhagen, 2004). For example, in the phrase “a goal was scored shortly after kick-off”, there is no explicit temporal information there, but there is some implicit information that could be obtained. In this case, the events of the goal being scored and kick-off are identified, and there is a temporal ordering between them, as well as implicit temporal information in these events themselves.

Much recent research has focussed on identifying and annotating temporal relations, a task that builds on top of temporal expression recognition and normalisation; however, a high performing temporal recognition and normalisation system is still required for this work to be effective.

## Annotation Standards

A number of standards for annotation of temporal expressions have emerged over time. The first annotation formats were typically based on SGML and XML and were simply in a format decided by the tagger. Over time, a standardisation effort for annotation emerged, culminating in TimeML (Pustejovsky, et al., 2003). TimeML is an XML-based annotation language, complete with a set of guidelines for timex annotation, based on the earlier TIDES standard (Ferro, Mani, Sundheim, & Wilson, 2001) and work in Setzer (2001).

Of most interest to this project in the TimeML specification is the TIMEX3 tag, which extends the annotation functions of the earlier TIMEX (Setzer, 2001) and TIMEX2 (Ferro, Mani, Sundheim, & Wilson, 2001) tags. An example of this tag is shown in Sample 1.

INDEPENDENCE, Mo. \_ The North Atlantic Treaty Organization embraced three of its former rivals, the Czech Republic, Hungary and Poland on <TIMEX3 tid="**t3**" type="**DATE**" functionInDocument="**NONE**" temporalFunction="**true**" value="**1999-03-12**">**Friday**</TIMEX3>, formally ending the Soviet domination of those nations that began after World War II and opening a new path for the military alliance.

Sample 1 - A sample TIMEX3 tag from the AQUAINT corpus (Verhagen & Moszkowicz, 2008)

The TIMEX3 tag is used to represent time expressions, and a number of attributes are used to define this. The most important attribute is the ‘value’ attribute, based on the TIMEX2 ‘val’ attribute, which is used to hold either the normalised time, or an unanchored duration. This value can be either a simple string referencing a specific time, a pair of strings separated by a slash representing a duration anchored in specific points of times, or a simple string representing an unanchored duration.

The format used for denoting dates is based on the modifications of ISO 8601 described in the TIDES standard (Ferro, Mani, Sundheim, & Wilson, 2001), with a number of modifications. As natural language temporal expressions allow a differing degree of precision, the TimeML standard allows for unknown components of a date to be replaced with the character ‘X’ (e.g., XXXX-05-03 represents May 3rd, when the year is unknown). Expression values are also omitted from right-to-left to the appropriate level of precision (e.g., 2010-05 for May 2010, but 2010-05-XX for ‘a sunny day in May 2010’).

To support further imprecision in natural language expressions that the ISO 8601 standard does not handle, TIDES, and subsequently TimeML, specify a number of replacement components which can be used as values in particular components of an ISO 8601 expression. This includes tokens such as “DT” in the hour position to represent “day time”, “WI” in the place of month to represent “winter” and “WE” in the place of a day to represent “weekend”.

In addition to these modified ISO 8601 values, a number of tokens are also allowed in the value attribute when expressions cannot be resolved to a timestamp, for example: “PRESENT\_REF” for time expressions such as “currently”; “FUTURE\_REF” for “future”; and “PAST\_REF” for “long ago”.

The second TIMEX2 attribute adopted by TIMEX3 is the MOD attribute, which is used for timexes that have been modified in natural language in such a way that cannot be expressed by value alone. These modifiers alter points in time and durations, allowing for expressions such as “before June 6th”, “less than 2 hours long”, or “about three years ago” to be correctly expressed.

TimeML’s TIMEX3 tag does not directly incorporate the other attributes of TIMEX2, but captures the information in other ways. One such attribute is the “functionInDocument” optional attribute that indicates whether this tag is providing a temporal anchor for other timexes in the document. The values this attribute can take come from the PRISM standard (IDEAlliance, 2008), and denote that a timex can take functions such as creation time, publication time, etc. The PRISM standard is typically used to mark up metadata to a document, rather than directly dealing with the content itself, whereas TimeML expands this to allow the content of the document to be tagged with these functions.

TimeML also allows a timex to be annotated as a “temporal function” (e.g., “two weeks ago”), and supplies a number of attributes to support the capturing of this data. Similarly, more attributes are provided to denote quantified times (such as “twice a month”), and to anchor durations to other timexes.

As interest in temporal expressions has grown to include event identification and temporal relations, the TimeML standard also includes tags and annotation guidelines for more than just timexes, such as events, signals for determining interpretation of temporal expressions and dependencies between these events and times.

In addition to the formal specification of TimeML, a set of annotation guidelines has been published (Saurí, Littman, Knippen, Gaizauskas, Setzer, & Pustejovsky, 2006), which contains information about when an expression should be tagged, and how the attributes should be filled, in order to ensure consistency between TimeML annotated documents. For the TIMEX3 tag, these are mostly inherited from the TIMEX2 guidelines, which are built on top of two basic principles (Ferro, Mani, Sundheim, & Wilson, 2001):

1. “If a human can determine a value for the temporal expression, it should be tagged.”
2. “VAL must be based on evidence internal to the document that is being annotated.”

The TIMEX2 guidelines then continue to specify a number of situations where a timex should be tagged, including detailed indicators of when to and when not to trigger a tag. One rule it gives relates to proper nouns, where any temporal expression incorporated within (e.g., the terrorist group “Black September”) should not be tagged, and a proper noun treated as an atomic unit. Additionally, specific rules are given to the extent of a tag, for example, when a temporal expression includes pre-modifiers (as handled by the ‘mod’ attribute), the pre-modifiers should be part of the tagged text.

The annotation guidelines for TIMEX2 also include guidelines for the format of the expected output tag (particularly for the form of the value attribute), depending on the type of expression that was recognised.

The TimeML rules extend these TIMEX2 guidelines, usually because of changes in the TIMEX3 tag from the TIMEX2 tag. These include changes in tagging extent recommendations for expressions embedded within each other, and for post-modifiers.

Additionally, TimeML also allows for empty TIMEX3 tags, which can be used to denote implicit timexes in text, often for anchored durations.

As with the wider TimeML standard, the annotation guidelines additionally define how to annotate events, signals, and relations; however as this project focuses on the annotation of timexes only, they are not considered here.

## Evaluating Tagger Performance

Contests for temporal expression recognition date back as far as the Message Understanding Conference of 1995, but only as part of a broader named entity recognition task. In 2004, the Automated Content Extraction (ACE) programme launched the Time Expression Recognition and Normalization (TERN) evaluation sub-task (MITRE, 2004), which focussed on two sorts of systems – those that perform recognition only, and those that perform both recognition and normalisation.

Although both TIDES (Ferro, Mani, Sundheim, & Wilson, 2001) and TimeML (Pustejovsky, et al., 2003) define annotation guidelines for the TIMEX2 and TIMEX3 tags respectively, the competitions also define additional guidelines which were used for the hand-tagging of the gold standard datasets. Issues with inter-annotator agreement were identified by (Setzer & Gaizauskas, 2001), so the purpose of these additional guidelines is to ensure high inter-annotator agreement.

The TERN contest defines system performance by using f-measures against different metrics of the system. An f-measure, sometimes referred to as an F1 score, is the harmonic mean of precision and recall. Precision is a measure of relevance – that is, of all the identified timexes or normalised values, what proportion of those are true positives or accurate. Recall is a measure of retrieval – that is, of all the possible timexes or normalised values in the document, what proportion of these were identified.

The first metric TERN uses to measure performance is that of detection of temporal expressions. The second is to recognise correctly the extent of the temporal expression, and the third is to normalise correctly the temporal expression into some time. This final metric also can be split into an absolute f-measure, which considers the performance of normalisation against all timexes, not just those recognised. Therefore, the absolute f-measure gives the headline metric for all parts of the system, whereas the breakdown allows performance of individual components to be considered. For the systems given below, we consider the recognition metric as both the recognition and extent detection tasks, and normalisation as the final, non-absolute metric.

Using these metrics, the best performing system for recognition is the ATEL system (Hacioglu, Chen, & Douglas, 2005), and for normalisation, Chronos (Negri & Marseglia, 2004). These systems, and others, are discussed further in section 2.5 below.

## Corpora

There are few publicly available corpora annotated with TIMEX tags. The TERN competition saw the creation of the TERN corpus, which consists of English and Chinese text annotated with TIMEX2 tags (Ferro, Mani, Sundheim, & Wilson, 2001). The texts that make up the TERN corpus are drawn from news articles. Performance on this corpus is typically used as a comparative measure between different systems.

The TimeBank corpus (Pustejovsky, et al., 2006) is a later corpus that extends the TERN corpus to use the TimeML standard (Pustejovsky, et al., 2003), also including additional documents, still from the news genre. Most recent contests use the TimeBank corpus as a base, although typically modify it for their specific needs (for example, in TempEval, a simplified form of TimeML was used).

A final corpus of note is the AQUAINT corpus (Verhagen & Moszkowicz, 2008), sometimes referred to as the ‘Opinion’ corpus, which also uses news texts and is annotated to the same specification as the TimeBank corpus, although the annotation effort is not as mature. Efforts are underway to merge the AQUAINT and TimeBank corpora into a new, larger corpus with a higher annotation standard.

The corpora discussed above are not considered perfect. As Setzer & Gaizauskas (2001) showed, high inter-annotator agreement on temporal expressions is hard to come by. In the case of the TimeBank corpus, inter-annotator agreement for TIMEX3 tags is 0.83 for exact matches, or 0.96 for partial matches (average of precision and recall). Other tags are lower, but they are of limited interest for this project and, as such, are not considered.

## Temporal Expression Taggers

Temporal expression taggers are tools that annotate the timexes in some input text. The earliest automated temporal expression annotation systems treated temporal expression recognition as a task along with entity recognition (Krupka & Hausman, 1998), and used simple hand-written rules (Mikheev, Grover, & Moens, 1998). In both systems, grammars were provided for the named entity recognisers and the time expressions simply recognised. No normalisation was performed in these early systems.

The recognition task is generally considered to be “do-able” (Ahn, Adafre, & de Rijke, 2005), with two main approaches to the task: rule-based and machine learning based. Unlike recognition, normalisation is considered a more difficult task, especially for underspecified temporal expressions, and durations.

Temporal expressions are recognised as being highly idiosyncratic, at least in English, but attempts have been made by linguists to make generalisations of the underlying grammar (Flickinger, 1996). Rule-based automated annotators use this principle by attempting to annotate timexes using these rule-based generalisations of the grammar.

Mani & Wilson (2000), in addition to the rule-based tagger discussed below, also experimented with machine learning based systems in order solve the problem of distinguishing between the specific use of the word “today” as a temporal expression and the generic use to mean “nowadays”. Following this, a number of machine learning based systems have been developed.

Machine learning systems generally all offer an advantage over other rule-based systems as the tedious creation of rules is avoided, and allows a certain amount of flexibility between languages. Some rule-based systems (Negri & Marseglia, 2004) maintain that in relatively short periods of time (i.e., one man-month) rule sets can be created which perform adequately. Negri & Marseglia (2004) also suggest that the coverage of rule-based systems can be easily extended by the simple addition of further rules, which can be simpler than improving the performance of machine learning systems.

With performance between machine learning and rule-based systems as close as it is there is no clear superior approach to timex annotation, with different authors extolling the advantages of their chosen approach.

The tasks of automated recognition and normalisation are often rolled into the same tool, although Ahn, Adafre, & de Rijke (2005) argues that separation of these components is beneficial. More recently, larger toolkits handling temporal expressions and relations have emerged (Verhagen, et al., 2005), where each component is modularised.

A number of temporal expression annotators are discussed further below.

### TempEx and GUTime

TempEx (Mani & Wilson, 2000) is a rule-based tagger that accepts a document tokenised into words and sentences and tagged for part-of-speech. A number of operations are applied to this input document, the first of which is the identification of the extent of the time expression. A number of regular expression rules are used to define the extent of what should be tagged.

The second module deals with the normalisation of self-contained expressions, and then a third module, called the “discourse processing module”, deals with relative expressions. For relative times, a reference time is established from the document creation date, and then rules handle temporal expressions representing offsets from this date by first computing the magnitude of the offset (e.g., “month”, “week”, etc.), and then the direction, either from direct indicators (e.g., “last Thursday”) or from sentence tense (“600,000 barrels were loaded on Thursday”).

GUTime (Verhagen, et al., 2005) is an extension to the TempEx tagger that extends the capabilities of TempEx to include the new TIMEX3 tag defined in TimeML, as well as some expressions not handled by TempEx, such as durations, some temporal modifiers, and European date formats.

When evaluated against the TERN data, GUTime scored an f-measure of 0.85 and 0.82 for TIMEX2 recognition and normalisation respectively (Verhagen, et al., 2005).

The GUTime program itself has a number of deficiencies that make extending this software difficult. The tagging aspects of TempEx are provided in a number of very large Perl functions that are driven by a Perl script. This is wrapped around by another Perl script and additional rules were added to the TempEx Perl module to create GUTime.

When incorporated into toolkits, such as TARSQI (Verhagen, et al., 2005), there is yet again another wrapper to fit this into the toolkit. These multiple levels of wrappers are code that hides issues due to the monolithic nature of the core TempEx code. In particular, there is a very heavy coupling between the higher level tagging logic and the actual tagging rules – a single function is used which contains all the rules and logic. Similarly, the second and third modules as outlined above are coupled into a single function.

This program structure makes adding or changing rules difficult due to the coupling between the rules and the logic itself, and makes analysis of the rules difficult.

### Chronos

Chronos (Negri & Marseglia, 2004) was a system created for the 2004 TERN evaluation that, like GUTime, provides one system for recognition and normalisation. However, these two tasks are split into separate internal components. Chronos is designed to be a multi-lingual system, coping with both English and Italian text.

One main difference between Chronos and GUTime is that Chronos can handle plaintext; tokenisation and part-of-speech tagging occurs in the first phase of the program. This does have the downside of making Chronos more difficult to componentise; if it were to be incorporated into a larger system, this pre-processing may want to be separated out to use a better system.

The recognition phase of Chronos uses about 1000 hand-written rules (considerably more than GUTime), which not only identify an expression and it’s extent, but are also used to collect information about an identified expression (such as modifiers and other “clues”) which help the later normalisation phase. Additional rules also exist which handle conflicts between possible multiple tagging. In GUTime, this is handled by an implicit rule ordering.

Additionally, Chronos, in contrast to GUTime that has a clear separation of components, appears to have a heavier coupling and a more integrated system. This recognition phase results in an intermediate representation – an extension of the TIMEX2 standard – that provides the metadata detected in the recognition phase as additional attributes to a tag.

Although this intermediate representation causes a heavy coupling between the two modules of Chronos, it may offer some advantages in reducing any repetition between the two modules by utilising all the information gleamed in the recognition stage.

Normalisation continues in a similar way to that proposed by (Mani & Wilson, 2000). Expressions are classified as either being absolute or relative, and then in the case of relative dates, the direction and magnitude of the relativity is determined and combined with a base date (determined in the recognition phase) to produce an anchor in time.

At TERN 2004, Chronos achieved the best results, with an f-measure of 0.926 and 0.872 for recognition and normalisation respectively, a performance that the authors put down to their more extensive rule set.

### DANTE

DANTE (Mazur & Dale, 2007) was a system submitted for the later 2007 TERN evaluation, again using the TIMEX2 schema.

Like Chronos, DANTE takes in plain text, so suffers from the same issue of componentisation as Chronos. Also similar to GUTime and Chronos, DANTE uses grammar rules (using the JAPE system) for identification of timexes. In this recognition phase, a “local semantic encoding” is used, which is an extension of the TIMEX2 standard that produces a (typically underspecified) value for the TIMEX2 ‘val’ attribute. The interpretation phase then takes this “local semantic encoding” and transforms it into a document-level encoding, using a number of assumptions on the progression of the timeline through the document.

Despite the different thought processes behind this (considering the semantics of a timex), the actual system is very similar at a high level to Chronos, yet an F-measure of only 0.7589 was achieved for TIMEX2 extent recognition, so performance is lower.

### ATEL

ATEL (Hacioglu, Chen, & Douglas, 2005) differs from the systems presented to this point in that it uses a machine learning approach to recognition, however does not handle normalisation at all.

ATEL takes full advantage of the machine learning approach to flexibility with languages by testing both Chinese and English, but different feature sets are required for both languages, and the results between them differ. The system scans for words as tokens, and then classifies each token as either ‘O’ for outside a time expression, or ‘(\*’, ‘\*’, or ‘\*)’ for the beginning, inside or end of a time expression respectively. The expected input to the system should be segmented into sentences and tokenised in order to facilitate this.

Each word is associated with a number of features in a sliding window, and a support vector machine classifier is used to classify tokens, expanding the possible classifications to include classes like ‘((\*’ and ‘\*)))’ to allow for embedded expressions.

At the 2004 TERN evaluation, the system scored an f-measure of 0.935 and 0.905 for TIMEX2 detection in English and Chinese respectively.

### TimexTag

TimexTag (Ahn, Rantwijk, & de Rijke, 2007) uses a machine learning approach, but unlike ATEL, also incorporates normalisation. Unlike the rule-based systems covered, TimexTag contains two distinct components for recognition and normalisation and concentrates on maximising performance of each component, rather than as an overall system.

Unlike ATEL, TimexTag does not identify timex phrases by considering the individual tokens, but treats it as a phrase classification task, by classifying each node in a parse tree as timex or non-timex. Again, support vector machines are used with a number of lexical and parse-based features.

Once these timexes have been identified, a classifier is used to categorise the phrases into the type of timex they represent semantically (e.g., recurrence, duration, a point in time, and the vagueness of these). Once again, a SVM is used for this classification, and the same features as in phrasal identification are used.

The TimexTag system is not based completely on machine learning, as rules are used to compute an under-specified representation for the start of the normalisation phase. However, these rules number considerably fewer than in other systems (89 vs. the 1000+ in Chronos). As with other systems, a base date, or “temporal anchor” is used to compute relative dates, and this is determined using simple heuristics. As with other systems discussed previously, the magnitude and direction of a relative timex also needs to be determined, which in TimexTag is once again using a SVM, utilising the same feature sets as before, but also considers tense of surrounding verbs as a feature (a similar approach to Mani & Wilson, 2000).

At the 2004 TERN evaluation, an f-measure of 0.899 was scored, although the absolute f-measure was lower.

### Rule Induction

An alternate machine learning approach to temporal annotation is that of rule induction. Baldwin (2002) presented a language-independent temporal expression annotation scheme that uses rule induction techniques to generate rules from an annotated corpus.

The rule induction method implemented here first attempts to classify the incoming TIMEX tags into types (durations, references, dates, and set-denoting expressions) and specificity (absolute/fully specified, relative/underspecified, and containing ‘X’ placeholders). Fully specified data is then processed separately, in order to discover a standard form for natural language expressions of dates that can be used with less specified expressions. The learning component then creates a regular expression with which to match the rule, and a set of instructions with which to evaluate the value.

This system obtained an f-measure score of 0.220 for recognition and 0.091 for normalisation, but this was against the French dataset, not the TERN dataset, so is not directly comparable to the other systems presented here.

Later work (Jang, Baldwin, & Mani, 2004) built on this with Korean text. Here, morphological analysis and a stop list are used to match temporal expressions in a text from a dictionary. Extending the annotator to include part-of-speech information and information about temporal modifiers is identified as a technique to build this dictionary automatically. Normalisation of temporal expressions is instead based on a rote-learning technique, where memorisation of relative expressions and their relative values is used, instead of attempting to generalise these as in (Baldwin, 2002). The scores here were considerably better, with an f-measure of 0.869 for normalisation against the Korean corpus.

# Problem Analysis

…

# System

The TERNIP system is implemented in Python (van Rossum, 1995) as a package called ternip. The package is distributed with an installer, documentation on how to use the package, and a series of extra scripts that demonstrate how to use the API, and provide a simple driver to tagging functionality.

As a core goal of this project is to deliver a high quality, re-usable tool that can be extended and used as a basis for further work, TERNIP was implemented following software engineering best practices. In particular, a continuous integration system (Fowler, 2006) was set up, and the principles of test-driven development (Beck, 2003) were followed. Combining high unit test coverage with continuous integration reduced the risk of bugs in the finished system, and allowed for a safe environment for refactoring to occur.

Porting the support functions from GUTime specifically benefited from this approach to development. As part of this translation process, functions were converted into equivalent Python, and unit tests written to check the ported functions behaved as would be expected. As the Perl functions had no unit tests themselves, the expected behaviour was as documented in the file. Once this had been done, some functions were then refactored to a more “Pythonic” way – for example, using Python library functions (particularly for time and date handling). Unit tests could then be used to verify that the refactored functions behaved as before.

Below, we discuss the high-level architecture of the system, followed by the implemented rule-based approach for timex recognition and normalisation and the document format wrappers.

## Architecture

The ternip package comes with two sub-packages: rule\_engine and formats, which each contain distinct components of the system. Additionally, the ternip package provides two functions: normaliser() and recogniser() which can be used to easily instantiate the “current best” normalisation and recognition components. At present, there is only one implemented module for both, but instantiating the recogniser and normaliser in this way allows improved techniques to be implemented later without any API changes to applications using TERNIP.

Also in the ternip package is a class, ternip.timex, which allows for abstract representation of the attributes of a time expression. The members of this class are inspired by the TIMEX3 attribute as described in TimeML (Pustejovsky, et al., 2003) and are documented fully in Table 1.

|  |  |
| --- | --- |
| Member | Description |
| id | A numerical identifier for the timex |
| type | A string indicating the type of expression annotated by this timex (this can hold any string, but some annotation schemas, such as TIMEX3, restrict the set of allowable values) |
| value | A string (in ISO 8601 basic format with the TIDES extensions (Ferro, Mani, Sundheim, & Wilson, 2001)) indicating the temporal value of the annotated expression |
| mod | A string indicating a modifier to the temporal value, e.g., to indicate that the value is approximate (this can hold any string, but some annotation schemas, such as TIMEX3, restrict the set of allowable values) |
| freq | A string in the format of a number followed by a character indicating the unit granularity (e.g., 3D to indicate 3 days) which indicates the frequency the expression reoccurs |
| quant | A string indicating how a value expression representing a set of dates should be quantified |
| comment | A string which can be used to add additional information to the timex (this is used by TERNIP during debugging to indicate the identifier of a rule which created or annotated the timex) |
| temporal\_function | A Boolean indicating whether or not the value needs to be determined via evaluation of a temporal function |
| document\_role | A string which indicates the role of the timex within the context of the document as an anchor for other timexes (this can hold any string, but some annotation schemas, such as TIMEX3, restrict the set of allowable values) |
| begin\_timex | When the annotated expression is a duration, this should hold the timex object which represents the start of the period covered |
| end\_timex | When the annotated expression is a duration, this should hold the timex object which represents the end of the period covered |
| context | The timex object which represents the temporal anchor for the annotated expression |
| non\_consuming | A Boolean which indicates if this timex represents an implicit time reference (i.e., one which does not consume any tokens) |

Table 1 - attributes on the ternip.timex class

A support function, add\_timex\_ids(timexes) is also provided, which annotates the id attributes in a set of timex objects so each timex has a unique identifier.

Internally, TERNIP represents documents as a list of sentences, where each sentence is a list of tuples, consisting of the token, the associated part-of-speech tags from the Penn Treebank tag set (Santorini, 1990), and a set of timex objects associated with that token. A sample of such a representation is shown in Sample 2, where a document consisting of a single sentence (“He derided Egypt for signing a peace treaty with Israel in 1979.”) with the penultimate token annotated as a timex. In the case where a timex spans multiple tokens, then the same timex object will be associated with every token in the expression.

[[('He', 'PRP', set()), ('derided', 'VBD', set()), ('Egypt', 'NNP', set()), ('for', 'IN', set()), ('signing', 'VBG', set()), ('a', 'DT', set()), ('peace', 'NN', set()), ('treaty', 'NN', set()), ('with', 'IN', set()), ('Israel', 'NNP', set()), ('in', 'IN', set()), ('1979', 'CD', set([<ternip.timex.timex instance at 0x058666E8>])), ('.', '.', set())]]

Sample 2 - A single sentence documented in TERNIP internal form

It is important to note that this representation represents a loss of fidelity from the original document because of the tokenisation process, specifically whitespace between tokens. However, the documented use of the classes in the ternip.format sub-package allows this internal format to be reconciled against the original document, meaning that this issue is avoided. This issue is discussed further in section 4.4.

In order to work with this internal format, a series of classes are provided which allows loading documents from disc, presenting them in the internal format, and for the internal format to be reconciled with the original document (for example, in XML documents adding the XML tags). These classes are discussed in further depth in section 4.4.

Once you have used a document class to get at the text in a file, you can then pass the internal format representation to the recogniser and then normaliser. Doing recognition by rule is discussed in section 4.2, and then normalisation in section 4.3.

## Recognition By Rule

The rule engine for recognition in TERNIP works by loading a list of rule objects and then checks for preconditions on the rules, which if passed, leads to the rule marking an extent of text within a sentence with a new timex. This is implemented in TERNIP as the ternip.rule\_engine.recognition\_rule\_engine class.

Rule preconditions come in two forms: ordering preconditions and ‘guards’. Other than the preconditions, a rule for recognition of timexes consists of a ‘matching’ regular expression, where the token sequence that matches this regular expression is defined as the extent of this timex, and a type definition, which indicates which type of timex this rule defines.

The rule engine passes the sentences one at a time to the rule class in the internal representation, however as regular expressions can only match against strings; this internal form must be converted into a string representation against which the regular expressions can match. This format extends on the interface given by the NLTK software (Bird, Klein, & Loper, 2009) in the nltk.text.TokenSearcher class.

In this representation, tokens are enclosed in angle brackets to indicate distinct tokens. The NLTK implementation, however, loses fidelity as part-of-speech tags and any currently associated timex objects are not included. Currently associated timexes are likely to be of limited interest for tagging; however, part-of-speech tags are a more interesting feature for determining timex extents and values. For this reason, the NLTK implementation is extended so that the token is appended with a tilde and then the part-of-speech tag for that token. Sample 3 shows the result of such a transformation on the sentence illustrated in Sample 2.

<He~PRP><derided~VBD><Egypt~NNP><for~IN><signing~VBG><a~DT><peace~NN><treaty~NN><with~IN><Israel~NNP><in~IN><1979~CD><.~.>

Sample 3 - a string representation as used for regular expression matching

The NLTK implementation also pre-processes regular expressions before compilation. This pre-processing results in cleaner regular expressions, as quantifying the . character will not result in an expression which matches across tag boundaries, i.e., a regular expression <.\*> will only match exactly one token, despite the greedy nature of the \* operator. This pre-processing introduces the restriction where expressions cannot match across word boundaries.

The second restriction is that angle brackets denoting token boundaries must be at the same bracketing level used for regular expression groups, e.g., the expression <mid(~.+><)?day~.+> will not perform as expected (that is, to match “midday” and “mid day” in one rule), due to the way the NLTK treats the token markers.

This pre-processing of regular expressions also allows for the introduction other conveniences. GUTime abstracted out common groups of words that appeared together to allow regular expressions to be shorter. TERNIP retains this functionality by replacing these pre-defined identifiers with some regular expression. These identifiers and their description are:

* $ORDINAL\_WORDS – ordinal values in word form (e.g., first, second, etc.);
* $ORDINAL\_NUMS – ordinal values in number form (e.g., 1st, 2nd, etc.);
* $DAYS – day names (e.g., Monday, Wednesday);
* $MONTHS – month names (e.g., February, December);
* $MONTH\_ABBRS – three-letter abbreviations of month names (e.g., Feb, Dec);
* $RELATIVE\_DAYS – relative expressions of day granularity (e.g., today, tomorrow, yesterday);
* $NTH\_DOW\_HOLIDAYS – holidays which always occur on the same day, in the nth week of a particular month (e.g., Labor Day, Mother’s Day);
* $FIXED\_HOLIDAYS – holidays which have a fixed date (e.g., New Year, Valentine’s Day);
* $LUNAR\_HOLIDAYS – holidays which are relative to Easter (e.g., Palm Sunday, Good Friday, etc.).

Additionally, rules can also be set to present strings that have number sequences marked up to allow for easy matching of such constructs. When this option is activated, the first number in a number sequence is preceded by the special identifier NUM\_START, and the final number in the sequence by NUM\_END, e.g., “twenty four” would be presented as NUM\_START<twenty~CD><four~CD>NUM\_END. Where the number sequence is an ordinal (e.g., “eighty fifth”), the markers NUM\_ORD\_START and NUM\_ORD\_END are used (e.g., NUM\_ORD\_START<eighty~CD> <fifth~CD>NUM\_ORD\_END).

An additional exception here is when the matching character . is immediately preceded by NUM\_START (or NUM\_ORD\_START) it will match the entire extent of the number sequence in a non-greedy fashion, but crossing word boundaries, which the NLTK pre-processing usually disallows.

Rules can also define a “squelch” option, which alters the rule into one that removes timexes from the matching extent and an option that defines whether regular expressions are case-sensitive (i.e., NUM\_START.\*NUM\_END will match a single number sequence, regardless of how many tokens it contains).

Rule execution by the rule engine proceeds one sentence at a time. On each sentence, the rule engine continually iterates the list of rules until all rules are marked as executed. As circular or dangling dependencies in the ordering precondition would mean this state is never reached (the ordering preconditions would always fail on these rules, leaving rules in the circle always stuck in the non-executed state in the rule list), then upon rule loading, these type of dependencies are checked for.

If the ordering precondition fails, then the rule is marked as not executed, left in the list for future iterations until its dependencies have been satisfied. Therefore, the largest number of iterations of the rule list needed is the size of the longest chain of dependencies.

Each rule can (optionally) have an ID, which can be referred to by other rules in an ‘after’ list – a list of IDs that this rule must be executed after. Execution does not have to be successful to satisfy this ordering precondition – a rule is marked executed when the preconditions of the rule are checked, regardless of whether those preconditions pass or fail. This is the ordering precondition explicitly defined in the rule. Rule blocks, discussed below, allow for specifying different ordering conditions implicitly.

The other set of preconditions to be considered is that of the ‘guards’: regular expressions which the sentence is matched against. These guards can either be positive, where at least one successful match is required to allow successful execution, or negative, where the regular expression must not generate any matches to allow successful execution of the rule. The first guard considered is a sentence-level guard, where the regular expression is matched against the whole sentence. Once this guard has passed, the ‘Match’ regular expression is applied to discover potential timex extents within the sentence. As a final step, two further sets of guards are checked before these extents are actually marked in the sentence. The first set of guards are the ‘before’ guard, where the token sequence preceding the extent of this match is checked, and the ‘after’ guards, where the token sequence proceeding the extent of this match is checked.

Following the success of all of these preconditions, then a new timex object is created, with the type indicated in the rule definition, covering the extent of the tokens that are matched. Because of this working on a token level, regular expressions are expected to belong to whole tokens, e.g., a regular expression which simply matches on the word “today” must contain the token delimiters: <today~.+>, otherwise the timex will not correctly annotate the whole timex.

### Simple Rule Files

Although rules the ternip.rule\_engine.recognition\_rule class allows rules to be created programmatically, an easier way to define rules is provided by loading and parsing text files containing rule definitions.

Each line starts with a key defining how the value should be considered, followed by a colon and then the value itself. Whitespace between the colon and the start of the value is disregarded, as well as trailing whitespace on the line. Additionally, rule files support comment lines. If a line starts with a single hash (#), then the rest of that line is ignored and considered a comment.

Rule file parsing is strict, and malformed rule files result in an error being raised by the rule engine and the rule failing to load. In order to aid with debugging, if multiple rules are being loaded at once (for example, from a directory of rules); the errors raised are delayed until all rules have been attempted to load, to give the most informative errors possible.

In each rule, a key can be specified once or multiple times, depending on the nature of what is being defined. Some keys can be omitted completely, in which case a default (or no) value is defined to that attribute of the rule. Table 2 lists the allowed keys in the definition of a recognition rule, and how their value relates to the rule descriptions defined above. Sample 4 demonstrates a minimal definition of a recognition rule.

|  |  |
| --- | --- |
| Key | Value Description |
| ID | The rule identifier: an optional string that can be defined no more than once, which can be referred to by other rules to express an ordering |
| After | The identifier of a rule whose execution must have preceded the execution of this rule: an optional string, that can be defined multiple times to define multiple dependencies |
| Type | The type of the expression identified by this rule: this is a string that must exist exactly once, and is assigned to the type attribute of the created timex object (see Table 1) |
| Match | The regular expression which, for each match in a sentence, results in a timex created covering the tokens matched in this expression: this is a compulsory field that can exist only once |
| Squelch | Whether to enable this rule as “squelching” rule: an optional Boolean (the strings ‘true’ or ‘false’), that can exist no more than once, and if omitted, defaults to false |
| Case-Sensitive | Whether the regular expressions should be matched case-sensitively: an optional Boolean (the strings ‘true’ or ‘false’), that can exist no more than once, and if omitted, defaults to false |
| Deliminate-Numbers | Whether number sequences should be marked up in the text before being presented to a regular expression: an optional Boolean (the strings ‘true’ or ‘false’), that can exist no more than once, and if omitted, defaults to false |
| Guard | A sentence-level guard for this rule: an optional regular expression that can exist multiple times, which results in a conjunction of conditions, or not at all, which results in an always-successful pre-condition. If the first character of the expression is an exclamation mark (!), then it negates the regular expression, meaning this guard will only pass if the regular expression does not match anything in the sentence. |
| Before-Guard | The before-match guard for this rule: this follows the same format as the ‘Guard’ key |
| After-Guard | The after-match guard for this rule: this follows the same format as the ‘Guard’ key |

Table 2 - Accepted fields in recognition rule definitions

Type: time  
Match: (<(about|around|some)~.+>)?<(noon|midnight|mid-?day)~.+>

Sample 4 - a minimal definition of a single recognition rule

In addition to this format, which allows one rule per file, the concept of “rule blocks” is also present. These blocks allow for an easier expression of ordering and for execution conditional on failure of all previous rules in the block.

Rule blocks follow a similar format to single rules, but allow for multiple rules in a file separated by three dashes: ---. Additionally, the first section of a rule block is the header of the rule block, which indicates the type of block it is. Sample 3 shows a sample rule block, and Table 3 shows the full list of acceptable values in the rule block header.

|  |  |
| --- | --- |
| Key | Value |
| Block-Type | A string of either ‘run-all’ or ‘run-until-success’ |
| ID | The block identifier: an optional string that can be defined no more than once, which can be referred to by other rule and blocks to express an ordering |
| After | The identifier of a rule or block whose execution must have preceded the execution of this block: an optional string, that can be defined multiple times to define multiple dependencies |

Table 3 - Accepted fields in rule block headers

As shown above, there are two types of blocks, which differ in how rule execution proceeds. In both types, rules are executed in order from top-to-bottom as defined in the file, but with the “run-until-success” type, execution of the block ceases when a rule executes successfully (the pre-conditions pass and the ‘Match’ regular expression results in at least one match). Any proceeding rules are not executed, and the rule block returns execution control successfully to the rule engine to continue its execution plan.

Because of the order implicit in a rule block, and that execution of a rule block proceeds atomically, explicit ordering of individual rules within the block is not permitted. Therefore, the ‘ID’ and ‘After’ keys, which are allowed in single rule files are not permitted in rules defined in rule blocks.

Block-Type: run-all  
---  
  
Type: date  
Match: (<(early|late)~.+>)?<last~.+><night~.+>  
  
---  
  
Type: date  
Match: <(early|late)~.+><(morning|afternoon|evening)~.+>

Sample 5 - a rule block containing two rules

### Complex Rule Files

For rules where the logic cannot be captured in the rule format described above (for example, more complex guards than regular expressions allow), an alternate rule format is supported, called complex rules. These complex rules are Python classes that implement a defined interface, and typically inherit from ternip.rule\_engine.rule, but are not required to do so.

Complex rules are written as Python files with the file extension .pyrule, which are imported by the rule engine, with the class named rule in the file instantiated and added to the list of rules.

These complex rules must implement two properties and one method to be compatible with the format required by the rule engine. These two properties are id and after. id is a simple string (or None), holding the rule identifier, and after is a list of strings (possibly empty) of rule identifiers which must have executed before this rule. The method that must be implemented is apply(sentence), which is called when this rule is to be executed. This function is called when this rule is to be executed, and is called with the sentence to be annotated, in internal format. Once annotation is complete, this method must return a tuple, where the first element is the annotated (or possibly unchanged) sentence in internal format, and the second is a Boolean indicating whether or not this rule executed successfully.

### The GUTime Rule Set

TERNIP provides with one rule set for recognition, which consists of rules translated from the GUTime (Verhagen, et al., 2005) software. These rules are largely regular expressions translated directly from the software by hand, with changes reflecting the differences in pre-defined constant substitution and the format for representing tokens. In cases where rule execution in GUTime is guarded by an if-statement, this is implemented as a guard.

There were three notable variations to this translation process that required three complex rules to be implemented. The first is that for year tagging, which in addition to before and after guards, will only tag a year if it is in a certain range (between 1649 and 2000), which would be unwieldy to express in a regular expression.

A second case for a complex rule is when tagging “the past”, where guards are required on the lack of presence of a timex starting in the same index. As mentioned above, the string representation of an internal format sentence does not capture this feature, so a complex rule must be written to work around it.

The final case is a special case for post-processing of the annotated expressions. It will merge adjacent timexes under certain circumstances, and removes over-tagged embedded timexes (a timex that is completely subsumed by another one).

To ensure accuracy of translation, this rule set was evaluated against GUTime’s recognition performance. This is discussed further in section 5.

## Normalisation By Rule

The normalisation rule engine has much in common with the recognition rule engine, a relationship captured by the ternip.rule\_engine.rule\_engine superclass, which both ternip.rule\_engine.normalisation\_rule\_engine and ternip.rule\_engine.recognition\_rule\_engine subclass. However, in order to satisfy the requirement of separation of recognition and normalisation, the rule engines are distinct components, only sharing some code where required (for example, rule loading).

The format of rules on disk is similar to in recognition rules, and is discussed further below. Similarly, preconditions on ordering and guards, and rule blocks behave as with recognition rules. To provide a consistent interface for writing rules, the internal format is transformed into a string representation for regular expressions in the same way, but this can be overridden to a simpler if convenient. Similarly, marking up number sequences (but only if the default representation for the internal format is used) and changing the case-sensitivity of regular expressions behaves identically to in recognition rules.

How execution of the normalisation rule engine precedes differs from recognition rule engine. Rules are executed on a per timex, rather than per sentence, level, however execution does proceed one sentence at a time (although not necessarily in the order timexes appear in the sentence). As execution proceeds, rules can also change the current “context” of the document – a base date/time that rules can use to compute relative expressions. At the start of execution, the current context is set to creation time of the document, which is passed in along with the document to the normaliser. This creation time does not have to be a complete date-time string, although many rules rely on at the date being specified to at least day granularity to operate correctly.

When a rule is executed, it is given the timex object to be annotated, a date/time string of the current temporal context of the document and of the creation time of the document. Additionally, the tokens (in internal form) which the timex object covers, the tokens preceding the timex extent in the sentence and the tokens proceeding the timex extent in the sentence are passed, which the rule uses to check its preconditions.

As with recognition rules, guards at a sentence-level, before the extent, and after the extent can be set to restrict successful execution of the rule. An additional guard against the timex extent can also be set. This is required as in recognition rules, the ‘Match’ field matches against the entire extent of the expression, however in normalisation rules, the ‘Match’ field can match a subset of the expression, so an additional guard against the entire expression is required.

In addition to these regular expression guards, a pre-condition for the timex type also exists

Another difference between normalisation and recognition rules is the role of the ‘Match’ field. In recognition rules, this is used to determine the extent of the timex to tag, however in recognition rules, the match groups defined in the regular expression are available to expressions which set the values of the timex attributes.

As mentioned above, an alternate representation of the internal form can be used against which regular expressions can match. This alternate representation does not contain part-of-speech information, but this is not necessarily considered by all rules so the optional omission is acceptable. In this format, tokens are simply joined by an optional delimiter. Sample 6 shows an example of such a representation, showing the simplicity of the detokenisation process, i.e., the space between the token “1979” and the full stop.

He derided Egypt for signing a peace treaty with Israel in 1979 .

Sample 6 - a single sentence using a simple string representation and a space separator

An additional restriction with the simpler string representation is that number sequences cannot be annotated, as this process requires for the token delimiters t be present.

If the ordering pre-conditions pass, and the rule is executed, then the guards are first checked, and then the ‘Match’ regular expression. If a match is found, then then Python expressions are executed (if the rule has one) and their values assigned to the appropriate attribute on the object.

These expressions are unrestricted in what they can execute; however, care needs to be taken to ensure they do not interfere with the execution of the rule engine, which would result in undesirable results. In order to help the normalisation process, a series of helper functions have been defined. These are discussed further in section 4.3.3.

These expressions are subject to a small amount of pre-processing to allow terser statements to be made. In order to access the matching groups of the ‘Match’ statement, the expression {#*n*} is used, where *n* refers to the number of the group in the expression. This is replaced with the Python code match.group(*n*) (where match is the match object resulting from the regular expression being executed).

As with recognition rules, all regular expressions for normalisation rules are pre-processed in the same way. This makes it important to remember that text such as $DAYS in the regular expression will be replaced with a match group for the days of a week, so when determining which match group to use in expression, these must be taken in to account.

### Simple Rule Files

As with recognition rules, the easiest way to load normalisation rules into the rule engine is via files on disk. These files follow the same format as the recognition rules, but with differing allowed fields and their meanings, corresponding to the description of normalisation rules given above.

|  |  |
| --- | --- |
| Key | Value |
| ID | The rule identifier: an optional string that can be defined no more than once, which can be referred to by other rules to express an ordering |
| After | The identifier of a rule whose execution must have preceded the execution of this rule: an optional string, that can be defined multiple times to define multiple dependencies |
| Type | A guard against the type of timex which this rule will normalise that is matched as a string case-insensitively: this is an optional string which can exist only once |
| Match | The regular expression that the extent of the timex must match for this rule to execute – the groups in this expression are exposed to the expressions below: this is a compulsory field that can exist only once |
| Guard | A guard against the timex extent for this rule: an optional regular expression that can exist multiple times, which results in a conjunction of conditions, or not at all, which results in an always-successful pre-condition. If the first character of the expression is an exclamation mark (!), then it negates the regular expression, meaning this guard will only pass if the regular expression does not match anything in the sentence. |
| After-Guard | The after-extent guard for this rule: this follows the same format as the ‘Guard’ key |
| Before-Guard | The before-extent guard for this rule: this follows the same format as the ‘Guard’ key |
| Sent-Guard | The sentence-level guard for this rule: this follows the same format as the ‘Guard’ key |
| Value | A Python expression (pre-processed as explained above) which is evaluated to set the ‘value’ attribute on the timex: an optional value which can exist no more than once |
| Change-Type | A Python expression (pre-processed as explained above) which is evaluated to set the ‘type’ attribute on the timex: an optional value which can exist no more than once |
| Freq | A Python expression (pre-processed as explained above) which is evaluated to set the ‘freq’ attribute on the timex: an optional value which can exist no more than once |
| Quant | A Python expression (pre-processed as explained above) which is evaluated to set the ‘quant’ attribute on the timex: an optional value which can exist no more than once |
| Mod | A Python expression (pre-processed as explained above) which is evaluated to set the ‘mod’ attribute on the timex: an optional value which can exist no more than once |
| Tokenise | How the internal format is represented as a string for regular expressions: if omitted or ‘true’ then the default representation is used, otherwise the value is used as the delimiter between tokens (the special strings ‘space’ and ‘null’ represent a single space and no delimiter respectively) |
| Deliminate-Numbers | Whether number sequences should be marked up in the text before being presented to a regular expression: an optional Boolean (the strings ‘true’ or ‘false’), that can exist no more than once, and if omitted, defaults to false |

Table 4 - Accepted fields in normalisation rule definitions

Type: date  
Guard: (last|past)  
Guard: !<(shrove|ash|good|palm|easter)~.+>  
Before-Guard: !<(the|a)~.+>$  
Match: <($DAYS)~.+>  
Value: compute\_offset\_base(cur\_context, {#1}, -1)

Sample 7 - a more complex normalisation rule for normalising expressions like “Last Tuesday”

These rules can also be structured into blocks of rules, as described in section 4.2.1, except where the allowed keys and values follow the format for normalisation rules described above. As with recognition rules, normalisation rules in a rule block differ from single ones in ordering conditions – ‘ID’ and ‘After’ are invalid in a rule block.

### Complex Rule Files

Complex rule files are also supported by the normalisation engine. These are files containing a Python class called ‘rule’ that are loaded by the rule engine, and the class instantiated and added to the list of rules in the rule engine.

These rules use the same attributes as for recognition rules to define the ordering pre-conditions: id and after. However, the function that is called when this rule is to be executed differs from recognition rules: apply(timex, current\_context, document\_creation\_time, body, before, after). This rule is expected to return a tuple consisting of a Boolean indicating whether it successfully executed and a date/time string containing the temporal context of this sentence (which this timex may have changed).

### Normalisation Support Functions

Many expressions contain patterns in the process they are converted to an ISO 8601 representation. Normal software engineering practice would encourage abstraction of these patterns into functions that can be reused, however the simple rule format does not allow for definition of Python functions. To allow these abstractions to be developed and their benefits to be realised, TERNIP provides the sub-package ternip.rule\_engine.normalisation\_functions that contains generic functions that rules can use.

Many of these functions were converted from GUTime (Verhagen, et al., 2005), tweaked to work in a more Pythonic way, but others were considerably expanded from their GUTime functionality, and more newly created.

These functions are classified into one of three classes: string conversions, date calculations, and relative date manipulations. An additional subclass of string conversion functions is also provided, which convert sequences of number-words (or ordinals) to their integer values. This follows the algorithm implemented in GUTime (Verhagen, et al., 2005), with an extension to support mixed integer and word sequences (e.g., “6 thousand”).

Other string conversion functions include ones that take a season name (e.g., ‘spring’) and return the corresponding identifier for use in a value field (i.e., ‘SP’), and date calculation functions include date\_to\_week, which converts a date to a week granularity string containing the week number (e.g., ‘2010W26’). These functions are documented using inline documentation.

The final set of functions relate to calculations of offsets from dates. Three functions are provided: offset\_from\_date, compute\_offset\_base and relative\_direction\_heuristic.

The first function takes a base date/time, the offset value, the unit of the offset (e.g., day, month, etc) and whether the resulting date/time should be of the original granularity, or the granularity of the offset made. Although this function was originally included with GUTime, it has been substantially rewritten to make it more resilient, and uses Python’s datetime module to provide much of the logic.

The compute\_offset\_base function will take simple relative expressions (such as ‘yesterday’, ‘Wednesday’ or ‘Easter’), a base date/time, and a direction hint to compute a new relative date. This can be used in non-trivial expressions such as “4 weeks from last Monday” to normalise the “Monday” phrase (with a negative direction hint) which can then be used as the base for an offset\_from\_date call, but it can also be used to normalise the trivial expression “last Monday” when it stands on its own. Like offset\_from\_date, this was originally ported from GUTime, but has been extensively extended to handle more expression types. The direction hint is used to determine the behaviour of this function, except where direction is implicit in the expression (i.e., ‘yesterday’ and ‘tomorrow’). When the direction hint is negative, it returns the date of first occurrence of expression before the reference date, even if the current date is an instance of that expression. e.g., if a negative direction hint is used, the reference date is the 25th December, and the expression is “Christmas day”, then the date returned will always be in the previous year. When the direction hint is positive, the behaviour is similar, except the dates will be the proceeding instance of that expression. When no direction hint is given, the closest instance of that date (e.g., if the reference date is a “Wednesday”, the expression “Tuesday” would resolve to the past, whereas “Thursday” would resolve to the future), is returned. Unlike when direction hints are given, it can return the same date as passed in, if that date is an instance of that expression.

The final function, relative\_direction\_heuristic, is an implementation of GUTime’s direction heuristic that returns the direction (if one can determined) of the temporal expression. It first looks at the section of the sentence between any preceding timex and this timex to identify a key verb, and if none are found the the proceeding section of the sentence and if this fails, the entirety of the sentence preceding this timex. If a verb is found, its tense is used to determine the direction, otherwise if the word immediately preceding the timex is “since” or “until”, this is used as a linguistic cue.

### The GUTime Rule Set

The normalisation rules implemented in TERNIP are all derived from GUTime (Verhagen, et al., 2005), but small tweaks have been made to capture some generalities between rules the original Perl did not capture. Unlike recognition rules, the implemented normalisation rules do not require the complex representation, which is largely due to the additional expressiveness allowed by executing Python code. This expressiveness can be used to guard against existing timex values, as Sample 8 shows (note that the entire Value expression should be on one line – the line breaks are not present in the rule definition).

Match: <night~.+>  
Value: (timex.value + 'TNI') if (re.match(r'^\d{8}$', timex.value if timex.value != None else '') != None) else timex.value

Sample 8 - adding approximate time-of-day expressions from gutime-timeofday.ruleblock

The Value expression in Sample 8 alters the value of an existing timex, but only if the current timex matches a defined regular expression (also guarding against type errors which is caused when the value is unset), or leaves it unchanged if not.

As rules are called multiple times on the same timex, ordering preconditions can be set to chain normalisation of different components in a timex, however the GUTime rule set only does this in one case.

Accuracy of translation from GUTime to the new TERNIP rule format was a key concern in implementing this rule set, and this was checked by comparing results from GUTime against TERNIP on the TERN data set to ensure they were identical. Although performance did differ on a small number of rules (largely due to improvements and generalities captured in the translation process), the results are similar. The system performance is discussed further in section 5.

## Document Formats

The restriction of TERNIP’s rule engines to operate on the internal format described in section 4.1 would significantly reduce the utility of TERNIP if no method was provided to parse documents into the internal format, and then to annotate the original documents with the new timex data. In order to combat this, a number of classes are provided in the ternip.formats package which implement this needed functionality.

As both the important TIDES and TimeML standards are implemented using XML, it is clear that support for the XML document format is needed. In addition, an implementation is provided of the annotation standard defined for use in the TempEval-2 competition (Pustejovsky & Verhagen, 2009), which borrows heavily from the TimeML standard, but is implemented on top of a standoff format.

The use of XML does present one downside. The TERN corpus (Ferro, Mani, Sundheim, & Wilson, 2001) is in SGML format, a superset of XML. However, XML was designed to provide a simpler version of XML, and Python support for SGML is lacking compared to its extensive XML library. Therefore, a decision was made to implement TERN format support with XML, restricting some documents in the corpus from being successfully parsed. This issue is discussed further in section 6.

One advantage XML has over plain text is that it can contain additional metadata about a document, or specific contents of it. This functionality is what is used to annotate the timex extents and provide the timex attributes, but is not limited to this. The document creation time is a key piece of metadata for the normalisation process, and many XML formats embed this in a document header, e.g., the DATE\_TIME attribute in TERN corpus documents. TERNIP is also particularly interested in sentence and token boundaries, and part-of-speech information, so if such information is available in a document, it is beneficial to use it. Sentence and token boundaries are determined by element nodes (e.g., in <sent>This is a sentence.</sent>, the element node is sent which contains a text node child of This is a sentence.), and part-of-speech information as an attribute on that node (e.g., <token partofspeech=”NNP”>TERNIP</token>). Additional metadata TERNIP concerns itself with are timexes, which are element nodes spanning the extent of that timex, with attributes on the element node. The exact format of the timex element depends on the specifics of the format being used (e.g., TIMEX2 for TIDES, TIMEX3 for TimeML). The document formats in TERNIP support loading all of this metadata from an XML document, and adding it to an XML file.

If sentence boundaries, token boundaries or part-of-speech tags are missing from the input document, TERNIP will use the NLTK (Bird, Klein, & Loper, 2009) to add it.

Supporting loading timex objects from documents in this way allows for TERNIP to run in just a normalisation role, where recognition is done by a third-party component, and also for conversion between types.

The Document Object Model (Apparao, et al., 1998) is a standard interface for manipulating XML documents by representing them as a tree, and this model is used by TERNIP in order to implement XML manipulations. Although loading documents from the tree is relatively straightforward, the act of reconciling the document with the internal format is harder, largely due to the whitespace lost during the tokenisation process, unless the sentence and token boundaries are tagged in the document.

For reconciliation in a document with no sentence or token annotations, the strings in the document must be aligned with the relevant tokens. The DOM tree is traversed depth-first (text nodes can only appear as leafs) to handle one text node at a time. In each text node, the offset of each token is determined by looking for the next occurrence of the first character of the token (starting the search from the end of the previously found token in the string). If that token is determined to be the start of a delimiter (i.e., a sentence, token or timex extent marker) then the text node is split at that point and a new node indicating the extent inserted in the split. The next step is to determine how far this node should extend. This is done by finding token extents as before and then splitting the text node again, and changing the cut-up text to the child or the newly inserted element. If the token extent is determined to be in a different text node, then the sibling nodes between the text node containing the start token and the end node are also moved to be the child of the new element node. If the start and end tokens are in text nodes that are not siblings, then the element node cannot be created, as that could not be represented in a tree without changing the order (and therefore meaning) of existing nodes. This situation is equivalent to an XML document that requires overlapping nodes (e.g., <tag1>This is <tag2>an overlapping</tag1> sentence</tag2>) which is an illegal XML representation.

Reconciliation and conversion for the TempEval-2 format is considerably easier, as it works on a per-token level, therefore the alignment and tree manipulations are not needed.

In order to implement the different XML formats supported by TERNIP: those with TIDES’ TIMEX2 tags and those with TimeML’s TIMEX3 tags, inheritance is used extensively. Figure 1 is a UML class diagram demonstrating the structure of this package. Classes that are more concrete also extend these, which allow for the construction of new documents from the internal format (with optional token offsets), and for extracting document creation time information, which is specific to individual formats.



Figure 1 - Structure of ternip.formats package

The ability to add sentence, token and part-of-speech tags to an existing XML document has utility beyond the scope of TERNIP. This ability was used to develop a wrapper for GUTime (Verhagen, et al., 2005). GUTime requires that TERN documents are marked up with sentence boundaries, token boundaries and part-of-speech data, but the TERN corpus does not contain this and it must be added in pre-processing, and then removed in post-processing before being given to the scorer.

Similarly, the ability to create documents from the internal format allows wrappers to be created for tools that require a specific input format. This ability was again used with GUTime, to allow it to be evaluated against the TempEval-2 corpus, by first converting to the TERN format and then back again.

The lack of token offset data in the internal format is problematic for document creation. During the detokenisation process, whitespace is inserted between tokens. Token offset data can be passed in separately to the internal format, allowing for correct construction of such a document, therefore working around this restriction, and avoiding data unnecessary for recognition/normalisation in the internal format. In the event token offset data is not necessary, a naïve approach is made to detokenisation, where a single space is inserted between every token.

# Evaluation

## Software Verification

Talk about unit testing

## Performance Evaluation

### Speed and Throughput

One key metric for evaluating system performance is that of speed of the system, both in terms of actual time, and in data throughput. To remove the overhead of the NLTK tokenisation and part-of-speech tagging routines, the subset of TERN corpus that can be parsed as XML (226 documents) was first marked up with sentence boundaries, token boundaries, and part-of-speech tags.

As performance against GUTime is a key concern for this project, GUTime was also evaluated in the same way against the same dataset. The pre-processing to add the sentence, token and part-of-speech information for TERNIP (to avoid NLTK overheads) is also required for GUTime, so it can be run directly on this pre-processed dataset.

A final concern for giving a fair result is that GUTime is likely to have substantial overhead due to the Perl interpreter and script having to be loaded per document, whereas using the TERNIP API eliminates have this overhead, as the library can stay in memory between multiple documents. To give a fairer result, the documents are passed to the TERNIP script, as well as annotated using the API, and compared to GUTime’s result.

Another, less realistic, method to discover throughput is create a single large file, and giving that to the taggers. This file was constructed by taking the file ‘20000715\_AFP\_ARB.0054.eng’ from the TERN corpus, and then repeating the content within the body of the document 100 times.

The entire system was also evaluated against the TERN dataset, including the tokenisation and part-of-speech tagging processes from the NLTK, both as part of the TERNIP API and as a wrapper around GUTime, which gives an indicator of real world performance, when handling unprocessed data.

The speed and throughput tests were repeated five times, and the mean taken, which is shown in Table 5. This experiment was performed on a modern PC with a 2.4 GHz Intel Core 2 Quad Q6600 processor, 4 GB RAM and Intel X25M hard drive, running Windows 7 x64, Python 2.6.5 and Strawberry Perl 5.12.0.

|  |  |  |
| --- | --- | --- |
| Multi-Document | Execution time (s) | Throughput (kbytes/s) |
| TERNIP (script) | 324.4 | 5.08 |
| TERNIP (API) | 50.64 | 32.51 |
| GUTime | 88.65 | 18.57 |
| Single Document |  |  |
| TERNIP (script) | 23.67 | 28.18 |
| GUTime | 15.65 | 42.63 |
| Multi-Document (including pre-processing) | | |
| TERNIP (API) | 146.0 | 2.124 |
| GUTime | 461.2 | 6.709 |

Table 5 - Performance of TERNIP and GUTime against the TERN corpus

# Discussion

Things to cover include:

* Criticism of GUTime – not really ported to TIMEX3 well (VAL vs. VALUE, etc)
* Perceived deficiencies in the TimeML spec, specifically the QUANT field, ambiguities in ISO spec: ISO basic/extended both allowed, also issuing with scoring software. Is P7D = P1W?
* Difficulties of making engine tag neutral because of differences in set notation between TIMEX2 and TIMEX3. TIMEX object very TIMEX3 oriented – a theoretical TIMEX4, or indeed completely different notation will be different
* Internal use of ISO8601 basic also problematic, perhaps a more abstract object would have been better (but possibly made rules more verbose)
* How corpora suck in terms of XML. TERN problematic particularly because it’s SGML, so XML parser sometimes struggles. Others have S, LEX, etc, tags, mixed in with TimeML which makes no sense… Would be good for TimeML to define a namespace so it can be sensibly used in other XML documents
* Overhead of script vs. API, etc – more likely NLTK
* Some rules error if there’s not DCT

# Bibliography

Ahn, D., Adafre, S. F., & de Rijke, M. (2005). Recognizing and Interpreting Temporal Expressions in Open Domain Texts. In S. N. Artëmov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, & J. Woods (Eds.), *We Will Show Them: Essays in Honour of Dov Gabbay* (Vol. 1, pp. 31-50). College Publications.

Ahn, D., Rantwijk, J. v., & de Rijke, M. (2007). A Cascaded Machine Learning Approach to Interpreting Temporal Expressions. *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics* (pp. 420-427). Rochester, New York, USA: The Association for Computational Linguistics.

Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Le Hors, A., et al. (1998). *Document Object Model (DOM) Level 1 Specification.* W3C.

Baldwin, J. (2002). *Learning temporal annotation of French news.* Washington, DC: Graduate School of Arts and Sciences, Georgetown University.

Beck, K. (2003). *Test-driven development: by example.* Addison-Wesley.

Bird, S., Klein, E., & Loper, E. (2009). *Natural Language Processing with Python.* O'Reilly.

Ferro, L., Mani, I., Sundheim, B., & Wilson, G. (2001). *TIDES Temporal Annotation Guidelines.* MITRE.

Flickinger, D. (1996). English time expressions in an HPSG grammar. *In Studies on the Universality of Constraint-Based Phrase Structure Grammars Gunji*, 1-8.

Fowler, M. (2006, May 1). *Continuous Integration.* Retrieved August 20, 2010, from http://www.martinfowler.com/articles/continuousIntegration.html

Hacioglu, K., Chen, Y., & Douglas, B. (2005). Automatic Time Expression Labeling for English and Chinese Text. *Proceedings of the 6th International Conference in Computational Linguistics and Intelligent Text Processing,* (pp. 548-559). Mexico City, Mexico: Springer.

IDEAlliance. (2008). *Publishing Requirements for Industry Standard Metadata.* IDEAlliance.

Jang, S. B., Baldwin, J., & Mani, I. (2004). Automatic TIMEX2 tagging of Korean news. *ACM Transactions on Asian Language Information Processing (TALIP), 3*(1), 51-65.

Krupka, G. R., & Hausman, K. (1998). IsoQuest Inc.: Description of the NetOwlTM Extractor System as Used for MUC-7. *In Proceedings of 7th Message Understanding Conference (MUC-7).*

Mani, I., & Wilson, G. (2000). Robust temporal processing of news. *ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics* (pp. 69-76). Morristown, NJ, USA: Association for Computational Linguistics.

Mazur, P., & Dale, R. (2007). The DANTE Temporal Expression Tagger. In *Human Language Technology. Challenges of the Information Society: Third Language and Technology Conference, LTC 2007, Poznan, Poland, October 5-7, 2007, Revised Selected Papers* (pp. 245-257). Springer-Verlag.

Mikheev, A., Grover, C., & Moens, M. (1998). Description of the LTG system used for MUC-7. *In Proceedings of 7th Message Understanding Conference (MUC-7).*

MITRE. (2004). *Time Expression Recognition and Normalization Evaluation*. Retrieved April 28, 2010, from http://fofoca.mitre.org/tern.html

Negri, M., & Marseglia, L. (2004). Recognition and Normalization of Time Expressions: ITC-irst at TERN 2004. *TERN 2004 Evaluation Workshop.*

Pustejovsky, J., & Verhagen, M. (2009). SemEval-2010 task 13: evaluating events, time expressions, and temporal relations (TempEval-2). *DEW '09: Proceedings of the Workshop on Semantic Evaluations: Recent Achievements and Future Directions* (pp. 112-116). Boulder, Colorado: Association for Computational Linguistics.

Pustejovsky, J., Castaño, J., Ingria, R., Saurí, R., Gaizauskas, R., Setzer, A., et al. (2003). TimeML: Robust Specification of Event and Temporal Expressions in Text. *IWCS-5, Fifth International Workshop on Computational Semantics*.

Pustejovsky, J., Verhagen, M., Sauri, R., Littman, J., Gaizauskas, R., Katz, G., et al. (2006, April 17). TimeBank 1.2. Philadelphia: Linguistic Data Consortium.

Santorini, B. (1990). *Part-of-speech tagging guidelines for the Penn Treebank Project. Technical report MS-CIS-90-47.* Department of Computer and Information Science, University of Pennsylvania.

Saurí, R., Littman, J., Knippen, B., Gaizauskas, R., Setzer, A., & Pustejovsky, J. (2006). *TimeML Annotation Guidelines Version 1.2.1.*

Setzer, A. (2001). *Temporal Information in Newswire Articles: An Annotation Scheme and Corpus Study.* PhD dissertation, University of Sheffield.

Setzer, A., & Gaizauskas, R. (2001). A pilot study on annotating temporal relations in text. *Proceedings of the workshop on Temporal and spatial information processing* (pp. 1-8). Association for Computational Linguistics.

van Rossum, G. (1995). *Python Reference Manual.* CWI Report CS-R9525.

Verhagen, M. (2004). *Times Between the Lines.* Waltham, MA, USA: Brandeis University.

Verhagen, M., & Moszkowicz, J. (2008, January). AQUAINT TimeML 1.0 Corpus. Brandeis University.

Verhagen, M., Mani, I., Sauri, R., Knippen, R., Jang, S. B., Littman, J., et al. (2005). Automating temporal annotation with TARSQI. *Proceedings of the ACL 2005 on Interactive poster and demonstration sessions* (pp. 81-84). Ann Arbor, Michigan: Association for Computational Linguistics.