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

This dissertation presents TERNIP (Temporal Expression Recognition and Normalisation in Python), a system for recognition of temporal expressions in text and normalisation of those expressions to a concrete date and time. TERNIP is modular and agnostic to output format, supporting both TIMEX2 (Ferro, Mani, Sundheim, & Wilson, 2001) and TimeML (Pustejovsky, et al., 2003) standards. Recognition and normalisation is implemented using a rule engine and rule set converted from the GUTime tool (Verhagen, et al., 2005), which scores an f-measure for recognition of 0.68 and 0.82 for normalisation against the TERN (MITRE, 2004) corpus, comparable to the performance of GUTime. This modular nature of TERNIP encourages the creation of future robust annotation modules.

ACKNOWLEDGEMENTS

I would like to thank my project supervisor, Dr Mark Hepple, for his invaluable advice and guidance in implementation of the TERNIP tool and the structure of this dissertation, and Lyndsey Redpath for her many hours spent proof reading the final report.

TABLE OF CONTENTS

1 Introduction 1
2 Background 2
2.1 Temporal Expressions 2
2.2 Annotation Standards 3
2.3 Evaluating Tagger Performance 6
2.4 Corpora 6
2.5 Temporal Expression Taggers 7
3 Problem Analysis 13
4 System 15
4.1 Architecture 15
4.2 Implementation Methodology 17
4.3 Abstract Representation Of Timexes 17
4.4 Abstract Representation Of Documents 18
4.5 Rule Engine 19
4.6 Recognition Rules 32
4.7 Normalisation Rules 33
4.8 Document Formats 35
5 Evaluation 39
5.1 System Performance 39
5.2 Speed and Throughput 40
6 Discussion 42
6.1 Meeting The Requirements 42
6.2 Implementation Issues 42
6.3 Standard and Corpora Deficiencies 44
6.4 System Performance 45
6.5 Future Work 46
6.6 Conclusions 47
7 Bibliography 49

1 INTRODUCTION

In this dissertation, TERNIP (Temporal Expression Recognition and Normalisation in Python), a system for recognition and normalisation of temporal expressions, is presented.

Temporal expressions are words and phrases which refer to some point in time (Ahn, Rantwijk, & de Rijke, 2007), and the distinct, but related, tasks of recognition and normalisation refer to the identification and resolution of these expressions to some standard format expressing a point (or set of points or duration) in time.

Interest in temporal expressions arises from their obvious utility, both within the wider fields of linguistics and philosophy, and as a task within the natural language processing field. Recognition and normalisation of temporal expressions in natural language text is clearly an important task for humans to be able to function in modern society (for example, correctly recognising and normalising the temporal expression "next Wednesday at 6pm" in the sentence "Shall we meet next Wednesday at 6pm?").

Temporal expressions in natural language are very rich and are often ambiguous (for example, in the phrase "midnight on Tuesday", as midnight refers to the instant between two days, it is unclear whether this refers to the time between Monday and Tuesday, or Tuesday and Wednesday).

In the field of natural language understanding, being able to handle temporal expressions in a similar way is clearly desirable, for example, in automatic summarisation of news texts, where the ability to construct a chronology of events aids is useful. Section 2.1 looks at temporal expressions in further depth.

Previous work into temporal expression recognition and normalisation has been done, including the definition of standards for annotation (section 2.2) and evaluation (section 2.3), the development of annotated corpora (section 2.4), and tools for automated annotation of these expressions (section 2.5).

Section 3 analyses the current state of the field and defines a number of requirements for TERNIP to fulfil in order to be a useful tool for annotation. Section 4 then discusses in depth the implementation of TERNIP to meet these requirements.

TERNIP is then evaluated (section 5) and the results, as well as issues arisen during the implementation of the project discussed in section 6.

This dissertation finishes with suggesting areas of future development (section 6.5) and drawing some conclusions (section 6.6).

2 BACKGROUND

2.1 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.

2.2 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.

2.3 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 the TERN competition 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.

2.4 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.

2.5 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.

2.5.1 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.

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.5.5 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.

2.5.6 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.

3 PROBLEM ANALYSIS

The literature survey conducted above, combined with the originally defined project scope and proposal, shows a number of issues to consider in the construction of a new tool. This allows for the definition of a number of requirements for TERNIP. The first requirements can be defined from the project scope:

1. A tool, in Python, capable of recognising and normalising temporal expressions in documents
2. An implementation of a rule engine for this tool, which uses the GUTime rules to implement recognition and normalisation

It is clear that there are a large number of research tools available to solve the problem of temporal expression recognition and normalisation, however these tools are implemented in a standalone fashion and do not integrate well into larger tool-chains. GUTime, for example, requires a large amount of Python wrapping to be integrated into the Tarsqi Toolkit (Verhagen, et al., 2005).

3. For an API implementation of the tool, to allow for integration into other toolkits

The current implementation of GUTime consists as modifications to, and then a wrapper around, TempEx (Mani & Wilson, 2000). A system that allows for separation of the rules from the application logic would make the system more extendable.

4. For a rule engine which separates the actual rules from the application logic

Ahn, Adafre, & de Rijke (2005) discusses the issue of separation of recognition and normalisation components, pointing to the benefits of mixing and matching components from different systems to produce an overall better system.

5. For recognition and normalisation components within the system to be separated
6. To allow for modularity of the recognition and normalisation components so that they can be replaced with other modules at a later date

A final issue identified is the size of annotated corpora, and the two standards used between them – the older TIMEX2 tag and the newer TimeML standard. The heavy linking of some tools to TIMEX2 has now rendered them obsolete without a substantial effort to bring them up-to-date.

7. Modular input/output components, with the annotation modules themselves being format agnostic.

It is also useful to define a limit to the scope of the implemented system. The system should focus only on the tasks of recognition and normalisation of temporal expressions, rather than the full range of tags within TimeML, or relations between events and timexes, etc.

4 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 for tagging functionality.

Below, a high-level overview of the system, how to use it, and the implementation methodology, is given. Following this, more thorough implementation details on the internal representation of timexes and documents is given, as well as implementation details about the rule engine. Finally, details on the implemented rule sets are given, finishing with implementation details about the supported document formats in TERNIP.

4.1 ARCHITECTURE

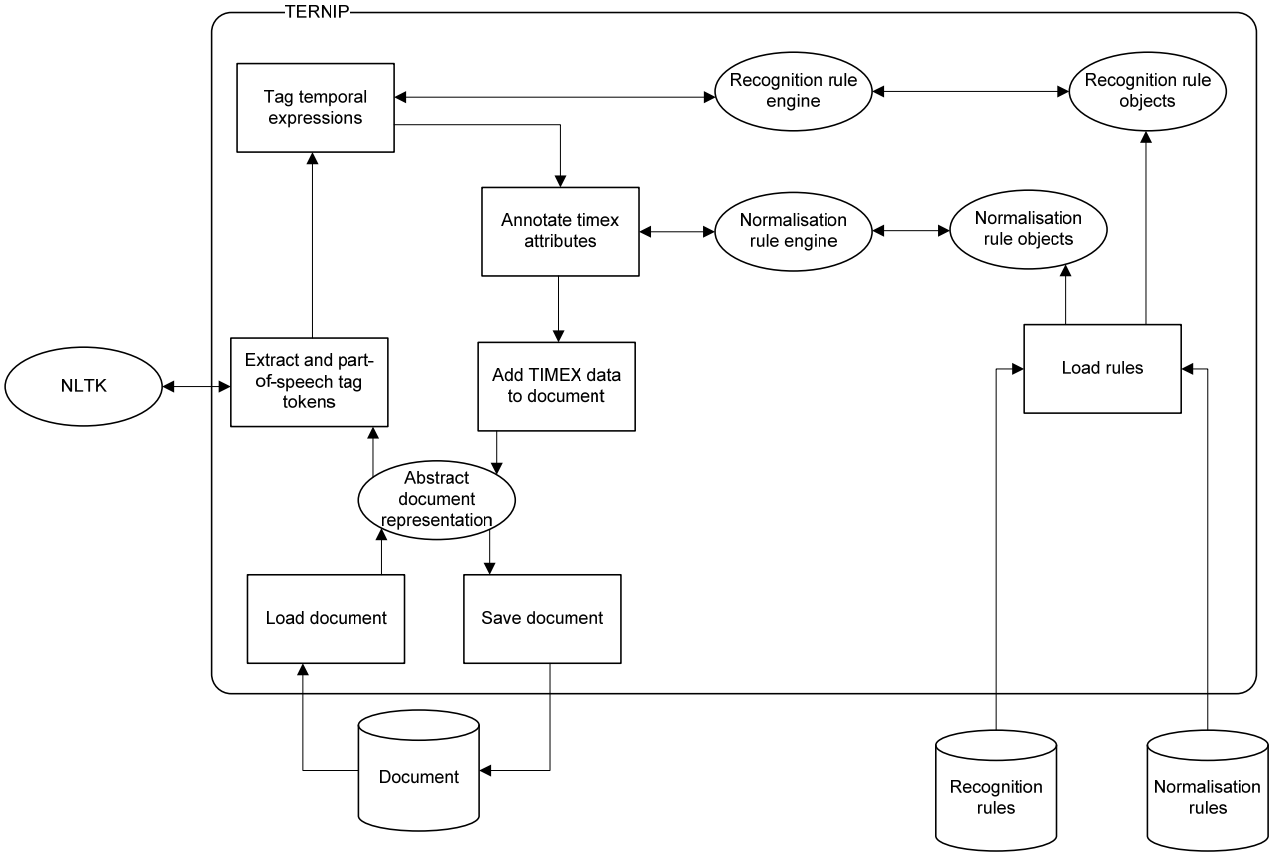

FIGURE 1 - HIGH LEVEL SYSTEM ARCHITECTURE AND DATA FLOW THROUGH THE RECOGNITION AND NORMALISATION PROCESS

Figure 1 shows a visualisation of the TERNIP system tagging a single document. In typical use, TERNIP will load a document into an abstract representation, then performs tokenisation and part-of-speech tagging using the NLTK (or metadata in the document). A list of tokens is provided to a recogniser, which identifies

temporal expressions (and their type) in the document, and then to a normaliser, which annotates the attributes (e.g., the value and any modifiers) of the identified timexes. Finally, this new timex data is added back to the original document representation, which can be converted back to a concrete representation and saved back to disk.

TERNIP is designed to be modular, so recognition and normalisation components can be removed and swapped with components that use different techniques, but in the current implementation of TERNIP, only one technique for recognition and normalisation is provided – a rule engine. These rule engines, one each for recognition and normalisation, load rule files from disk and execute these rules one sentence at a time for recognition rules, and one timex at a time for normalisation rules.

TERNIP provides a very simple interface for programs to use in the form of a Python package called `ternip`, containing two sub-packages: `rule_engine` and `formats`, which each contain distinct components of the system. For stand-alone use, a script called `annotate_timex` is provided, which further simplifies the use of TERNIP.

In order to further hide implementation details, 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 (the rule engine), but instantiating the recogniser and normaliser in this way allows improved techniques to be implemented later without any API changes to applications using TERNIP.

This technique is demonstrated in Sample 2 below, which shows the simplicity of the TERNIP API: loading the recogniser and normaliser, opening a document, tagging the expressions in the document, annotating the attributes to the timex, and then adding the timex data back to the document, and writing the document back to disk.

```
import ternip
import ternip.formats

recogniser = ternip.recogniser()
normaliser = ternip.normaliser()

with open(file_path) as fd:
    doc = ternip.formats.tern(fd.read())
sents = recogniser.tag(doc.get_sents())
normaliser.annotate(sents, file_creation_time)
doc.reconcile(sents)
with open(file_path, 'w') as fd:
    fd.write(str(doc))
```

4.2 Implementation Methodology

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.

4.3 Abstract Representation Of Timexes

In order to fulfil the requirement that TERNIP is agnostic to timex format, an abstract representation for timexes needs to be provided for internal use, with conversion to and from the target representation in the document to allow loading timex data from a document, or adding it to one.

This internal representation is supplied by a class (`ternip.timex`), the members of which 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 for output representations. Internally referencing timexes can be done by direct reference to the class, so IDs are only relevant when converted to an external representation; therefore, all IDs can be added at once, just before the external representation is made. External representations are discussed in further depth in section 4.8.

4.4 Abstract Representation Of Documents

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.

```
[[('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 3 - A SINGLE SENTENCE DOCUMENTED IN TERNIP INTERNAL FORM

A sample of such a representation is shown in Sample 3; 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.

It is important to note that this representation results in 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 is discussed further in section 4.8.

In order to work with this internal format, a series of classes are provided which allows loading documents from disk, converting them to the internal format for the recogniser and normaliser, 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 4.8.

4.5 Rule Engine

Two rule engines are implemented in TERNIP, one for recognition and another for normalisation, although much code and functionality is shared between the two.

In each case, the rule engine is responsible for loading rules from rule files on disk (the differing rule formats are discussed in sections 4.5.2, 4.5.3, and 4.5.5) and then executing these rules in an order such that any ordering preconditions (discussed in section 4.5.4) are satisfied.

When a rule is executed, control is passed to the rule, and each rule then checks whether the pattern matching conditions (section 4.5.1) defined by the rule are satisfied. If the conditions are satisfied, the rule applies the required action (for example, annotating a token extent as being a timex, or setting attributes on a timex object), before returning control back to the rule engine to execute the next rule. If the conditions are not satisfied, the rule returns control back to the rule engine and makes no changes to the document. This is discussed in further detail in section 4.5.6.

4.5.1 Pattern Matching

Successful rule execution is conditional on one pattern matching the sentence (recognition rules) or timex body (normalisation rules); however, additional pattern matching conditions, called guards (which must be satisfied), can be set.

Regular expressions are a standard notation for expressing pattern-matching rules; however, they can only match against strings. As the internal

representation of a sentence is a list of tuples, some transformation is required to allow this format to be represented as a string.

The NLTK class `nltk.text.TokenSearcher` (Bird, Klein, & Loper, 2009), solves a similar problem of building a string from a list of tokens. This method was used as a basis for the representation of the internal format list of tuples as a string. In this representation, tokens are enclosed in angle brackets to indicate token boundaries.

If just the NLTK class were used, fidelity between the internal format and the string would be lost, 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 the part-of-speech tag for that token. Sample 4 shows the result of such a transformation on the sentence illustrated in Sample 3.

```
<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 4 - 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 is implemented by replacing the `.` character with `[^>]`. This pre-processing introduces the restriction in which 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.

Extending this pre-processing of regular expressions also allows for the introduction of other conveniences. GUTime abstracted out common groups of words that appeared together to allow regular expressions to be shorter, as these groups could be included by using Perl variables. TERNIP retains this functionality by replacing some 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.).

The $ character was chosen due to its familiarity as an indicator for variable substitution, and also as its use of an end-of-string marker in regular expressions, means that no valid expression can have text following it.

Another convenience for rules is that in the conversion from internal format to a string representation number words can be marked up, which allow regular expressions to match these sequences in a much simpler way. 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 represented 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).

To allow for rule regular expressions that match any valid number sequence, the expression NUM_START.+NUM_END can be used. This is a markedly simpler expression than one that match number sequences directly. This expression does contradict the restriction above, where the . character will not match across token boundaries. One alternative would be the express this as NUM_START(<.*>)+NUM_END, but this has issues with greedy matching. In order to allow a compact expression, then when the . character follows the delimiter NUM_START (or NUM_ORD_START), its meaning is changed to not cross a NUM_END (or NUM_ORD_END) boundary, as opposed to the standard token boundary restriction, i.e., when . follows NUM_START or NUM_ORD_START, . is replaced with (?:.(?!NUM_START)).

4.5.2 Basic Rule Format

Rule engines are responsible for loading rules as text files from disk and creating rule objects from them. These rule objects are responsible for checking the conditions on rule execution, and then applying the rule action if these conditions are satisfied.

Rules on disk are expressed in text files, ending with the extension .rule, and on each line, a key defining the meaning of that line, and a value, which is interpreted depending on the key. Sample 5 shows a sample recognition rule.

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

SAMPLE 5 - A MINIMAL DEFINITION OF A SINGLE RECOGNITION RULE FOR PHRASES LIKE "AROUND MIDDAY"

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.

Both recognition and normalisation rules can have guards – regular expressions that must be satisfied for successful execution of the rule to take place. These guards can also be negative guards, where a successful match would block execution of the rule. Additionally, attributes relating to ordering and dependencies (discussed further in section 4.5.4) can be set.

The final condition on execution is that of the 'Match' attribute of the rule. The attribute role varies between recognition and normalisation rules. In recognition rules, the tokens matched by this pattern are annotated as a timex (there can be multiple matches in a sentence), whereas in normalisation rules, the groups in the regular expression are available to later normalisation expressions.

Rules also have a number of other options that change the action of the rule, or how pattern matching is executed. For both recognition and normalisation rules, the 'Case-Sensitive' option changes whether regular expressions are applied in a case sensitive manner, and the 'Deliminate-Numbers' option indicate whether number sequences in the sentence should be marked up as described in section 4.5.1.

Recognition rules can also define a 'Squelch' option, which alters the rule's behaviour to remove timexes from the matching extent, rather than add new ones; and a type definition, which sets the type attribute of the annotated timex.

Table 2 lists the keys allowed in the definition of a recognition rule, the value format, and meaning, related to the rule structure explained above.

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

Sample 6 below demonstrates a more complex case of a recognition rule, which uses the 'squelch' option to remove any overtagged expressions. This rule matches durations in the form of a number followed by a unit identified (e.g., "six years"), using marked-up number sequences for matching. Guards are then used

to restrict the rule further by considering the words before and after the matched extent.

```
Type: duration
Match: (<\d+~.+>|NUM_START.*NUM_END)<$UNITSs?~.+>
Deliminate-Numbers: True
Squelch: True
After-Guard: ^(<since~.+>|<after~.+>|<following~.+>|<later~.+>|<earlier~.+>|
<before~.+>|<prior~.+><to~.+>|<previous~.+><to~.+>)
Before-Guard: !<(the|for|in)~.+>$
```

SAMPLE 6 - A MORE COMPLEX NORMALISATION RULE DEMONSTRATING A RANGE OF OPTIONS

The acceptable keys for normalisation rules differ from recognition rules to reflect the different rule attributes that need to be set. Sample 7 demonstrates a simple normalisation rule, which returns a value (of week granularity) of the current date to normalise the expression “this week”.

```
Type: date
Match: <this~.+><week~.+>
Value: offset_from_date(cur_context, 0, 'W')
```

SAMPLE 7 - A SIMPLE NORMALISATION RULE FOR THE EXPRESSION "THIS WEEK"

Normalisation rules allow for overriding the conversion of the internal format into a string representation by a ‘tokenise’ option, which if set to anything other than true, uses a naïve method of detokenisation (joining tokens with some token delimiter). 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 8 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 8 - 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 token delimiters to be present.

This simpler string representation was added to make conversion of some rules from GUTime easier (as some normalisation rules do not consider tokenisation), in order to simplify the written regular expressions.

Normalisation rules allow for setting another condition on the rule execution, which is matched (case-insensitively) against the type attribute of the timex. Unlike other guards, this is a simple string matching, not a regular expression.

The final set of attributes unique to normalisation rules is that of Python expressions, the results of which, when evaluated, are set to the relevant timex attribute.

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 behaviour. In order to help the normalisation process, a series of helper functions have been defined. These are discussed further in section 4.7.1.

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). In addition to this, other variables are available to these expressions: `timex`, `cur_context`, `dct`, `body`, `before`, `after`, and `senttext`. These represent the timex object being normalised, the current date/time context of the sentence, the document creation time, and internal format tokens for the timex body, the preceding tokens and the following tokens, and the string representation of the body tokens.

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.

Table 3 below gives the full listing of allowed keys for normalisation rules, and their meanings, related to the rule description 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 3 - ACCEPTED FIELDS IN NORMALISATION RULE DEFINITIONS

Sample 9 below demonstrates a more complex normalisation rule using the guards above. It will normalise relative day expressions with a negative direction,

like “last Tuesday”, but with guards to protect against normalising dates related to Easter (which typically include the name of a day, e.g., Easter Sunday), or where the word before the timex is “the” or “a”.

```
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 9 - A MORE COMPLEX NORMALISATION RULE FOR NORMALISING EXPRESSIONS LIKE “LAST TUESDAY”

4.5.3 Complex Rule Format

For rules where the logic cannot be captured in the rule format described above (for example, more complex guards for a recognition rule 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. Sample 10 shows an example of a complex recognition rule. This rule creates a timex covering the phrase “the past”, but only when “the past” is not already tagged at the start of another timex.

```
import re
import ternip
import ternip.rule_engine.rule

class rule(ternip.rule_engine.rule.rule):
    """
    Special case of "the past" - this needs to come after the durations
    processing so it doesn't conflict with cases like "the past 12 days"

    Translated from GUTime
    """
    id = 'gutime-past'
    after = ['gutime-durations']

    def __init__(self):
        self._rule = re.compile(self._prep_re(r'<the~.+><past~.+>'), re.I)

    def apply(self, sent):
        senttext = self._toks_to_str(sent)
        success = False
        for match in self._rule.finditer(senttext):
            ti = senttext.count('<', 0, match.start())
            tj = senttext.count('<', 0, match.end())
            # Check that there isn't a TIMEX that already starts
            # with the same phrase
            guard = False
```

```
            ts = sent[ti][2]
            for t in ts:
                if ti == 0 or t not in sent[ti-1][2]:
                    guard = True
            if guard:
                continue
            # This rule succeeded
            success = True
            t = ternip.timex(type='date')
            self._set_timex_extents(t, sent, ti, tj, False)
        return (sent, success)
```

SAMPLE 10 - A COMPLEX RULE FORMAT THAT CREATES A TIMEX BASED ON THE PRESENCE OF OTHER TIMEXES

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.

For recognition rules, the method that must be implemented is `apply(sentence)`. This function is called when this rule is to be executed, 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.

For normalisation rules, the method that is called on execution is `apply(timex, current_context, document_creation_time, body, before, after)`. The arguments here are the timex object to be annotated, a date/time string representing the current temporal context and the document creation time, and then lists of tokens (in internal format) of the sentence parts forming the body of this timex, preceding the timex and following it. 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).

4.5.4 Ordering and Dependencies

As rule execution alters the document, or the timexes within it, rules may perform different actions depending on in which order they are executed. In order to reduce this ambiguity, rules can give themselves identifiers that can then be referred to by other rules to establish a relationship between the two rules. This is implemented in the form of an `id` attribute on a rule, and an `after` list. The `after`

list is a list of identifiers that must have been executed before this rule, i.e., a rule must be executed after the rules in its `after` list.

This ordering pre-condition is enforced by the rule engines during execution, and is discussed further in section 4.5.6. In the case where no explicit ordering is set, there is no guarantee on the order in which other rules will be executed.

Circular or dangling dependencies in the ordering precondition would result in rules that are unexecutable, as the ordering preconditions would always fail, causing issues for the rule engine. To avoid this, when rules are loaded, checking occurs against these types of dependencies.

4.5.5 Rule Blocks

In addition to simple and complex rules shown above, there exists the concept of rule blocks. Rule blocks group many rules together, with an implicit execution ordering, and the entire block is ordered as if it were a standard rule. Another key feature of rule blocks is to allow for conditional execution of rules based on the successful execution of previous rules in the block.

```
Block-Type: run-all
---

Type: date
Match: (<(early|late)~.+>)?<last~.+><night~.+>

---

Type: date
Match: <(early|late)~.+><(morning|afternoon|evening)~.+>
```

SAMPLE 11 - A RULE BLOCK CONTAINING TWO RULES

Rule blocks (of which there is one block per file) follow a similar format to single rules, but allow for multiple rules in a file separated by three dashes: `---`. The first section of a rule block is the header of the rule block, which indicates the type of block it is and its ordering conditions. Sample 11 shows a sample rule block, and Table 4 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 4 - 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 successfully executes (the guard conditions pass and the 'Match' regular expression results in at least one match). Following rules are not executed, and the rule block returns execution control successfully to the rule engine to continue its execution plan. This is equivalent to a condition on all rules in the block that restricts execution if a previous rule in the rule block has executed successfully.

Because of the order implicit in a rule block, explicit ordering of individual rules within the block, either to other rules in the same block, or to completely different rules, is not permitted. Because of this, rule block execution happens atomically. The 'ID' and 'After' keys, which are allowed in single rule files, are therefore not permitted in rules defined in rule blocks.

4.5.6 Flow of Control

Once the rule engine has been created, and the rules from disk loaded, execution starts by the rule engine being passed a document, and in the case of the normalisation rule engine, a document creation time. The rule engine is responsible for ensuring ordering, which is done in the same way in both recognition and normalisation rule engines.

Execution for each sentence or timex starts with all rules being added to a list. This list is then continually iterated until it is empty. As execution proceeds, the ordering pre-condition is checked as each rule is reached, and if this is satisfied, then the rule is executed and removed from the list, and the ID of that rule being added to a list of executed IDs. This occurs even if execution was not successful (e.g., a guard condition failed), i.e., the rule engine considers a rule executed when it fires the rule. Satisfaction of the ordering pre-condition is done by checking that all the IDs referred to in the 'after' list of a rule are in the list of executed IDs. If this is not true, the rule is not executed and left in the list for a future iteration. Therefore, the largest number of iterations of the rule list needed is the size of the longest chain of dependencies.

An alternative implementation of this would be to pre-sort the rule list into an order which reflects the partial ordering created by the 'after' list, however, for ease of implementation and the negligible overhead of checking for set membership in the current implementation, this is not implemented.

The rule engine treats rule blocks as if they were a single rule, and once execution is passed to the rule block, the rule block executes its constituent rules in order, and then returns control to the rule engine once it has completed execution, or reached the stopping condition.

Further details on how the rule engine proceeds for basic recognition and normalisation rules are below. With complex rules, the rule engine proceeds with execution in the same way, but once control is passed to the rule, the complex rule is in complete control of its own execution.

Recognition Rule Engine

For recognition rules, execution proceeds one sentence at a time, executing all rules (in order) on one sentence, and then continuing to the next sentence (if there is one) and running all the rules again. Once control is passed to the rule, the 'guard' condition is checked.

The guard is a list of regular expressions against which the sentence is matched. 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 check, 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 following the extent of this match is checked.

Following the success of all of these conditions, 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.

If all conditions are met, and the match regular expression matches something, then this rule is considered to have executed successfully.

Normalisation Rule Engine

For normalisation rules, execution of rules is on a per-timex, rather than per-sentence, level. As the normalisation engine proceeds through the document, it does so on a per-sentence level, and in each sentence, the timexes in the sentence are discovered, and the rules executed one timex at a time. It is important to note that execution order is not guaranteed to be in the order the 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 in against which conditions can be checked and expressions can use, if need be..

Once control is passed to the rule, guards at a sentence-level, before the extent, after the extent, and in the body of the timex are checked. The timex body guard is a condition not present in recognition rules, but is useful for normalisation rules. 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, requiring an additional guard. The final guard to be checked is that of timex type, which is not a regular expression, but a simple case-insensitive string equality check.

Once these conditions are checked, the normalisation rule match expression is executed, and if a match is generated, the Python expressions evaluated and their value assigned to the timex attributes.

4.6 Recognition Rules

TERNIP provides one rule set for recognition, consisting of 72 rules, which consists of rules translated from the GUTime (Verhagen, et al., 2005) code. These rules were 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 conditional on an if-statement, this is implemented as a guard.

It was found that almost all 'rules' in GUTime could be translated into the above format. There were three exceptions that could not, which required representation using the complex rule format. The first is 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 2100), 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 absence 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 test 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.

4.7 NORMALISATION RULES

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. There are 260 normalisation rules.

There are many similarities in the methods ISO 8601 representations are generated by different rules. 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. These normalisation support functions are discussed in further detail in section 4.7.1 below.

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 12 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 12 - ADDING APPROXIMATE TIME-OF-DAY EXPRESSIONS FROM GUTIME-TIMEOFDAY.RULEBLOCK

The `Value` expression in Sample 12 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 multiple rules can be called 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.

4.7.1 NORMALISATION SUPPORT FUNCTIONS

In order to allow for simpler value statements, as well as to take advantage of the benefits abstraction brings, a number of functions were written which Python expressions in normalisation 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 robust, 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 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 extended extensively in order to handle more types of expression. 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 next 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 is found 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.

4.8 Document Formats

The restriction of TERNIP's rule engines to operate on the internal format described in section 4.4 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 implements 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>`).

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.

Additional metadata TERNIP concerns itself with are timexes, which are represented in XML as 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 (i.e., `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.

Supporting the loading of 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 for conversion between types.

Once a document has been loaded, and the internal representation modified by the recognition and normalisation components, the changes to the internal representation (chiefly the addition of timex data, but also optionally tokenisation and part-of-speech data) must be merged back in to the document. This process is called reconciliation.

The DOM, or 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 through querying the DOM, 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 2 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.

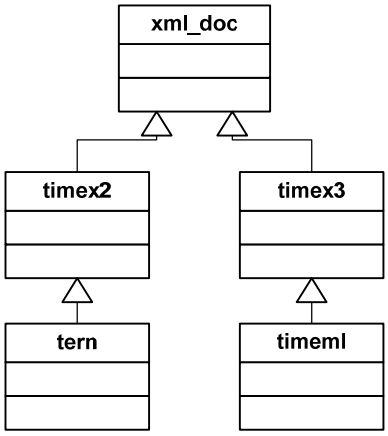

FIGURE 2 - STRUCTURE OF TERNIP.FORMATS PACKAGE XML DOCUMENT CLASSES

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 fully accurate 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 available, a naïve approach is made to detokenisation, where a single space is inserted between every token.

5 EVALUATION

The effectiveness of the implemented system is an important factor that needs to be considered. Below, the performance of the system in terms of the results of the temporal expression annotation process is evaluated, followed by a look at the speed of the system.

5.1 SYSTEM PERFORMANCE

The metrics of precision and recall are ubiquitous throughout natural language processing and are an effective measure of system performance, and are introduced in section 2.3.

The TERN (MITRE, 2004) and TempEval-2 (Pustejovsky & Verhagen, 2009) contests both provided sample gold standard corpora and a scoring mechanism for TIMEX2 and TimeML annotation respectively. These corpora and tools can therefore be used to evaluate the system, and give meaningful results to be compared against other systems.

In order to satisfy the requirements that the system gives similar performance to GUTime, GUTime was also evaluated in the same experiments.

The TERN corpus used is the ACE 2004 corpus, which consists of SGML documents. However, as TERNIP (and the pre-processing wrapper developed for GUTime) can only handle XML, only the subset of the corpus that can be parsed as valid XML is considered.

The TERN scorer outputs three relevant metrics for consideration with this project: recognition, extent, and normalisation. The recognition f-measure does not consider whether the extent of the `TIMEX2` tags differs, only if there is some overlap between a timex in the gold standard and a timex in the hypothesis file (e.g., tagging “Monday” would still score as a match, even if the full timex in the gold standard is “last Monday”). The extent f-measure is also a score of recognition performance, but with the harsher condition of the extents of the tag being identical.

The final metric, the normalisation score, considers the `val` attribute on the `TIMEX2` tag, giving a true positive if the strings match. The normalisation metric is given as the accuracy, i.e., the proportion of correctly annotated `val` attributes to the number of TIMEX2 tags in the gold standard that have `val` attributes. The TERN scorer was modified to convert the `val` attributes in both the gold standard and file to be scored into ISO 8601 basic format from extended, if need be. This issue is discussed at further length in section 6.

Although the TERN scorer outputs f-measures for each of these measures directly, simply taking a mean of these values results in macroaveraging effects (skewing

the score towards correctly tagging documents with few timexes). To avoid these effects, the true positive, false positive, and true negative numbers are taken directly and then the f-measure computed over the whole result set – microaveraging.

For the TempEval-2 evaluation, the training data set was used, and the scorer was adjusted similarly to the TERN scorer to convert ISO 8601 extended representations into their basic equivalent.

Normalisation scoring for TempEval-2 proceeds as in TERN (a test of string equality between the `value` attributes), but recognition scoring takes a different approach. The recognition score works on a per-token basis, rather than a per-timex and per-extent basis. Whether each token is included in a timex in both the gold standard and the file to be scored is used to determine how that tag should be classified. This results in a metric which gives partial credit for incomplete extent recognition, however does result in the score being skewed on expression length; missing one 10-token timex will be penalised in the same way as missing ten 1-token timexes.

An additional issue was identified with GUTime. In some circumstances, GUTime will introduce unbalanced TIMEX3 tags to a valid XML document, making it invalid for parsing back to the TempEval-2 annotation format. TempEval-2 scoring works on the entire corpus at once, not on a per-document basis as with TERN, and as documents it corrupts cannot be loaded back in, the scorer sees it as if GUTime tagged nothing in those documents, giving GUTime an artificially lower score. In order to address this, documents that GUTime corrupts were removed from the corpus to give comparable results.

TERN evaluation	TERNIP	GUTime
Recognition (any overlap)	0.68	0.68
Recognition (extents match)	0.57	0.56
Normalisation (`val` attribute)	0.82	0.57
TempEval-2 evaluation		
Recognition (per-token)	0.78	0.75
Normalisation (`value` attribute)	0.69	0.65

TABLE 5 - TERNIP AND GUTIME PERFORMANCE SCORES (MICROAVERAGED F-MEASURES)

These results are analysed in section 6.4.

5.2 SPEED AND THROUGHPUT

Another 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 this overhead, as the library can stay in memory between multiple documents. To give a fairer result, the documents were passed to the standalone TERNIP tagging script, as well as annotated using the TERNIP API (persistent in memory), and the GUTime script.

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 6. 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	5.075
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	6.709
GUTime	461.2	2.124

TABLE 6 - PERFORMANCE OF TERNIP AND GUTIME AGAINST THE TERN CORPUS

These results are analysed in section 6.4.

6 DISCUSSION

This section explores various issues of the implemented system and its performance. Section 6.1 analyses the system against the requirements identified in section 3, and section 6.4 analyses the performance of the system shown in section 5. Section 6.2 discusses issues identified during implementation of the system, and section 6.3 expands on this to discuss issues within the wider field of temporal expression recognition and normalisation.

To conclude the report, section 6.5 identifies aspects of future work that the system could be applied to, or which would improve the tool, and section 6.6 draws some high-level conclusions about the entire project.

6.1 MEETING THE REQUIREMENTS

An important issue to discuss is whether the implemented system meets the seven requirements defined in section 3.

The first two requirements are clearly satisfied by the system; the tool is written in Python, and does perform the task expected of it, via use of a rule engine and a rule set converted from GUTime. Additionally, as this is implemented in the form of an API, the third requirement is satisfied.

The basic rule format (section 4.5.2) encapsulates rules in a standalone representation, completely independent of the application logic, satisfying requirement 4. Further encapsulation of the differing application logic and differing rule formats for the recognition and normalisation components, as well as the modular architecture of TERNIP satisfy requirements 5 and 6.

The final requirement, number 7, is partially satisfied by the `ternip.formats` class and the internal representation, which allows for support for different document types, as well an abstract internal representation. Issues due to fundamental differences between TIMEX2 and TIMEX3 set notation made implementing an agnostic system difficult, along with other issues, which are discussed in section 6.2.

With the exception of minor issues with the satisfaction of requirement 7, TERNIP can be considered to implement the identified requirements successfully.

6.2 IMPLEMENTATION ISSUES

Although one of the core requirements was to support multiple different tagging formats, there are deficiencies in TERNIP's implementation of the TIMEX2 tag format. Due to the desire to represent timexes internally in an abstract way, and the fundamental differences between TIMEX2 and TIMEX3's handling of sets of dates, annotating particular attributes in TIMEX2 format is not possible. Another TIMEX2 attribute that was omitted from TERNIP is the GRANULARITY attribute.

This omission was made as the TimeML standard deemed it unnecessary, as granularity is implicit in the timex value.

Additional issues may arise if theoretical future standards need to be supported that represent a large shift from the fundamental ideas behind TimeML. The attributes on the timex object in TERNIP are almost all directly taken from the TIMEX3 attribute list, with a few small changes (for example, referring to other timexes by reference to the object, than an identifier). Similarly, as normalisation rules directly set timex attributes directly to an acceptable (or a heavily related representation) TIMEX3 form, support for future standards may require new fields or changes to normalisation rules (if automatic translation between two attribute value formats is not possible).

A similar concern with future proofing stems from the internal use of the ISO 8601 basic representation, with the TIDES extension. If a future standard uses a different format for representing dates, then translation will be required from the internal form to the new representation, complicating matters. One potential solution would be to use an abstract representation for date values, perhaps based on Python's `datetime` class, but with support for partial values and differing levels of granularity. However, this would have added significant complexity to the implementation, as well as making the expressions for setting the value attribute more complex (building a string is simpler than creating objects).

The translation process from GUTime to TERNIP also flagged up a number of deficiencies (clear typos in regular expressions, sections of rules with faulty guards that would never trigger, etc.) in GUTime, which were corrected in TERNIP. This is reflected in the marginally higher performance of TERNIP than GUTime on the same data.

Other issues were identified in GUTime, which made the translation to TERNIP rules harder. Most of these issues arose from GUTime's extension of TempEx (Mani & Wilson, 2000) to add TimeML support. These included set expressions, where GUTime still uses TIMEX2 annotation formats, rather than TIMEX3, which needed an almost complete rewrite of these rules for TERNIP.

A final implementation issue in TERNIP is that in some circumstances, normalisation rule expressions will generate errors. This is occurs when the document creation time is not set, in which case relative dates which use this as a base date will fail, and generate a warning. These warnings are harmless, but ugly, and future work may be necessary, perhaps to add guards on the current context date for normalisation rules.

6.3 STANDARD AND CORPORA DEFICIENCIES

The implementation of TERNIP flagged a number of issues with corpora and implemented standards.

The TimeML annotation guidelines define the QUANT attribute as "generally a literal from the text that quantifies over the expression" (Saurí, Littman, Knippen, Gaizauskas, Setzer, & Pustejovsky, 2006), which is unspecific and limits the usefulness for later processing, as the possible values for the field are not well defined. Future revisions to the TimeML specification would benefit from a clearer specification of valid values or the format of the QUANT attribute.

Other issues in annotation arise from the ISO 8601 standard and differing levels of tool support. Scorers, for example, often consider normalisation to be performed correctly if value attributes between a key and response timex are identical. However, different expressions can resolve to the same temporal meaning, e.g., 'P7D' and 'P1W' are temporally equivalent, as a duration of 7 days is equal to a duration of 1 week. Scorers that only consider string equivalence would mark down a response for this. Other tools may have similar issues.

Another issue with ISO 8601 is the support for both basic and extended (which allows separators for date and time components) formats. This leads to a situation similar to durations where two different strings are temporally equivalent, e.g., 2010-08-20 in extended format is 20100820 in basic format.

Future revisions to the TimeML standard may benefit from defining a restricted subset of valid ISO 8601 to reduce ambiguity. Another option would be to implement an abstract date/time class (as discussed above) which tools could use in order to provide robust parsing and operations on ISO 8601 date values.

A final issue that was observed during implementation of TERNIP stems from the format of TimeML corpora. TimeML defines a number of valid tags in its XML schema, however the TimeBank corpus (Pustejovsky, et al., 2006) contains a large mix of tags (such as denoting tag boundaries) which are not defined in the XML schema. This is semantically invalid XML. The XML specification does contain a method for defining multiple acceptable XML schemas (for example, one for sentence boundary annotation in addition to TimeML) through the namespacing scheme. However, with no recommendation in the TimeML specification as to what the recommended URI for namespacing should be, this is unreliably implemented (TimeBank uses the URI 'TimeML1.1.xsd', whereas AQUAINT uses the URI 'http://timeml.org/timeMLdocs/TimeML_1.2.1.xsd'). At present, TERNIP does not implement support for TimeML namespaces and can support considering sentence/token boundary tags not part of the TimeML specification.

The AQUAINT corpus (Verhagen & Moszkowicz, 2008) better implements the TimeML specification as the XML documents do not contain superfluous tags,

resulting in XML documents that are semantically well-defined. As effort continues to mature the TimeBank corpus and to merge it with AQUAINT, moving towards semantically well-defined XML should be a consideration.

Furthermore, future revisions of the TimeML specification should include a URI recommendation, which allows for tools that work with TimeML to support mixed format documents where TimeML is just one namespace within the document.

6.4 System Performance

6.4.1 Annotation Performance

Section 5 performed a series of tests to evaluate the performance of the system in the recognition and normalisation of temporal expressions, the results of which are shown in Table 5.

The performance of TERNIP is as expected for recognition on the TERN corpus. Small improvements in this task compared to GUTime do exist due to bug fixes made to the GUTime rules as they were converted to TERNIP rules, as well as other small changes to capture generality in some expressions. In the TempEval-2 corpus, the improvements over GUTime are larger; however, there is no clear reason for this. One possibility is that the conversion from TempEval-2 format to TERN format introduces spaces between all tokens as no token offset data is provided in the TempEval-2 corpus, where they may not be any expected by the GUTime rules.

Normalisation shows a different picture, with TERNIP performing considerably better than GUTime on the TERN data set. However, on the TempEval-2 set, this difference is lessened. The differences in TempEval-2 normalisation results could be put down to the small changes in functionality for normalisation and increase in generality captured during the rule translation process, but this would not explain the large discrepancy on the TERN corpus. Analysis of why GUTime performs substantially worse on the TERN dataset reveals that although normalisation is attempted for expressions which are incorrect, the resulting values are very far off the mark – in one case normalising the expression ‘today’ to ‘20301008’ when the document creation time is 15st August 2000. This suggests that the discovery of document creation time is flawed (as ‘today’ normalisation should just return the document creation time).

Further analysis of the problem discovered that the GUTime script only considers the `DATE_TIME` element in a TERN document to discover the document creation time, whereas some documents (those it fails on) uses the `DATE` element. When GUTime cannot determine the document creation time, it falls back to the current date, in the form YY/MM/DD (i.e., in the flawed case above, ‘10/08/30’) as the base for relative expressions. This issue is compounded, as the TempEx core

normalises this as an expression in the form MM/DD/YY, resulting in an extremely erroneous value, and causing the issue seen above. TERNIP's normalisation performance on the TempEval-2 corpus is the same as GUTime's performance as reported in Verhagen, et al. (2005), therefore it seems reasonable to consider that TERNIP performance is at least as good as GUTime.

Another important aspect to note is that the performance of GUTime against the TERN corpus is given by Verhagen, et al. (2005) as f-measures of 0.85 for recognition with any tag overlap and 0.78 for recognition with correct extents. There is no clear reason why these results could not be reproduced, but there are a number of possible reasons, including issues with pre-processing (for example, differing token boundaries or incorrect part-of-speech tags), or that GUTime performs particularly well on the non-XML parts of the TERN corpus.

6.4.2 Speed and Throughput

One disappointing aspect of TERNIP is demonstrated in terms of its overall throughput compared to GUTime. Profiling execution reveals no single bottleneck in TERNIP code; therefore, it seems that the different, more complex, architecture (such as manipulating the DOM of a document rather than treating it as a string) as well as different interpreter overheads, is what results in this discrepancy. In terms of real-world usage when tagging multiple documents, the benefits TERNIP brings through its API and memory persistence allow for benefits to be reaped, giving an approximately 40% increase in throughput.

Tagging multi-document sets that are not pre-processed is slower still; however profiling indicates that the part-of-speech tagging process in the NLTK is the bottleneck here.

However, this loss in speed is outweighed by the improvements in code quality and the encapsulation and decoupling of many components of the system, which lend TERNIP its key feature as a framework for future development. Regardless, there retains scope for further optimisation if required, for example using a faster part-of-speech tagger, as well as the benefits more detailed profiling and optimisation would bring.

6.5 Future Work

The implementation of TERNIP allows for its use as a framework to support the development of further recognition and normalisation modules, therefore there remains much possibility for implementation of existing techniques as TERNIP modules, or to support the development of new modules, such as ones which use machine learning techniques. Using TERNIP in this way allows implementation of new techniques to focus on the core issue of recognition or normalisation, distanced from the peripheral (but important) concerns of input/output representations, etc. The inclusion of scoring scripts and sample drivers in

TERNIP also gives a very low barrier for entry and to assist development – being able to score system performance from a very early point.

An additional aspect of future work with TERNIP is to use it as a component for integration into toolkits that perform temporal expression recognition and normalisation. Ease of integration was a key concern when developing the API, and if toolkits standardise around the TERNIP API, then these toolkits can be very easily changed to support new, improved, tagging techniques easily.

As well as future development to extend and integrate TERNIP as a whole within larger projects, there also remains a large amount of scope for improvement of the rules currently implemented. A rule discovery tool, which outputs rules in TERNIP's basic format based on expressions missed, or incorrect normalisation, compared to the gold standard would expand the rule set provided with TERNIP. Similarly, a tool that automatically compares the performance of TERNIP rules against different TERNIP rule sets (for example, checking if a new rule increases performance) would also aid development.

Unsurprisingly given the requirements, TERNIP contains a lot of potential for expansion, and the hope is that the NLP community find it useful for assisting in tasks of temporal recognition and normalisation.

6.6 Conclusions

TERNIP can be considered a successful implementation of the requirements to solve the problem of a robust, reusable tool (and framework) for temporal expression recognition and normalisation. Some minor issues regarding abstracting timex representations and extendibility exist, but they do not impact TERNIP's current performance.

Issues arisen during the implementation of TERNIP highlight that further work is required in order to develop more mature corpora and annotation standards. Deficiencies in corpora, particularly TimeBank (Pustejovsky, et al., 2006), exist in the implementation of the TimeML XML schema (Saurí, Littman, Knippen, Gaizauskas, Setzer, & Pustejovsky, 2006), compounded by under-specification in the TimeML standard. These issues can be addressed by defining a recommendation for an XML namespace URI in the TimeML standard, and for this to be implemented in TimeBank, leading to a corpus with semantically well-defined XML documents. Further ambiguity in the definition of TimeML fields, specifically the QUANT attribute, also would benefit from further work in future revisions of TimeML.

TERNIP as a whole allows for modularity in different components, lending itself well to future extensions to functionality and the introduction of new techniques for recognition and normalisation, as well as integration into wider toolkits. The

separation of rule definition from the rule engine leads to a highly maintainable and robust system.

TERNIP's rule engine component, with the rules derived from GUTime, scores an f-measure of 0.68 for recognition of timexes (when extents are not necessarily equal) and 0.82 for normalisation of the value attribute of the TIMEX on the TERN corpus. Compared to Chronos (Negri & Marseglia, 2004), which scored the highest f-measures at TERN 2004 of 0.926 and 0.872 for recognition and normalisation respectively, TERNIP's performance is clearly not ground-breaking, as a result of the rule set being converted from GUTime. However, as Chronos is also rule-based, an investigation into converting these rules to TERNIP's format may yield beneficial results.

Regardless, these results are positive, but illustrate that the problem of recognition and normalisation of temporal expressions is one that is not yet solved, and further work is needed. TERNIP's role as an extensible tool can assist further work for this problem.

7 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.