Leadership Trait Analysis and Threat Assessment with *Profiler Plus*

Nick Levine
Ravenbrook Limited
PO Box 205
Cambridge, CB2 1AN
United Kingdom
ndl@ravenbrook.com

Michael Young
Social Science Automation, Inc
3798 Dayspring Dr.
Hilliard OH 43026
USA
michael@SocialScience.net

ABSTRACT

Profiler Plus is a general-purpose “natural language” analysis application implemented in Common Lisp. We discuss its capabilities in the context of two points of view which are brought together here for the first time: that of Social Science Automation which commissioned the product and programmed the rules which drive it; and that of Ravenbrook which, without ever wholly understanding what it did, successfully implemented it.

Categories and Subject Descriptors

I.2.7 [**Artificial Intelligence**]: Natural Language Processing—*Text analysis*; D.3.2 [**Programming Languages**]: Language Classifications[Specialized application languages]; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*; I.5.4 [**Pattern Recognition**]: Applications—*Text processing*; J.4 [**Social and Behavioral Sciences**]: [Sociology]

General Terms

Human Factors

Keywords

Common Lisp, Political Science

1. INTRODUCTION

1.1 Content Analysis[1]

Profiler Plus[2] is a child of the late 1990s and of approaches to the assessment-at-a-distance of individuals and groups that originated in the analysis efforts of the British and American intelligence services of World War II. These were later influenced by advances in psychology and remain in use today by academics and governments alike. By the mid-80s many different approaches existed, under such diverse titles as: operational code, image analysis, leadership trait analysis, cognitive mapping, conceptual/integrative complexity, motivations, and more. However due to its labor-intensive nature this work was not flourishing: in each form of content analysis, every document had to be read at least once by a trained analyst, processed by hand according to some set of *rules*, marked up (*coded*) – annotations in the margin! for example `+D` might be used to indicate “distrust” – and then the data analyzed and interpreted. Problems abounded because the rules used by the human analysts were usually poorly specified and made huge reliance on the competence of native language users. As a result, the process was slow and plagued with reliability problems. As a practical matter the analysis produced was often too little, too late.

By the 1990s some efforts were being made to use computers to automate this process, but they largely relied on word or phrase lists which did not adequately capture the desired content and often ignored issues like negation of meaning, relying purely on term frequency counts. Machine learning was typically not considered, because of its relative inaccessibility to social scientists, the lack of transparency in how items are coded, and the *a priori* need for a relatively large and reliable training corpus.

A notable exception to this characterization was the KEDS/TABARI system developed for event coding by Dr. Philip Schrodt[3]. Unfortunately this system was tuned only for event coding and not suitable for general-purpose social science text analysis.

Social Science Automation (SSA) was founded in 1997 in part to meet this challenge and to provide a general-purpose platform that could embody any and all of the existing assessment-at-a-distance coding schemes (such as that discussed in Sect. 1.2) along with any others that might be devised. Its efforts drew the attention of the United States Department of Defense and, with their support, development of *Profiler Plus* began in earnest in the summer of 1998.

1.2 Example – Leadership Trait Analysis

A *coding scheme* is a set of *rules* for identifying constructions in text that have some meaning. Initially all of the schemes developed for Profiler Plus were for the at-a-distance assessment of foreign leaders and groups. Here we

[1] Holsti, O. R. *Content analysis for the social sciences and humanities* 1969. Don Mills: Addison-Wesley.

[2] `http://socialscience.net/tech/ProfilerPlus.aspx`

[3] `http://eventdata.parusanalytics.com/`

discuss one aspect of one such scheme: “Leadership Trait Analysis”[4].

Leaders who are high in distrust of others are given to being suspicious about the motives and actions of others, particularly those who are viewed as competitors for their positions or against their ideology or cause. These others can do nothing right; whatever they do is easily perceived as for ulterior motives and designs. In its extreme, distrust of others becomes paranoia in which there is a well-developed rationale for being suspicious of certain individuals, groups, or countries. Distrust of others often makes leaders not rely on others but do things on their own in order to prevent any sabotage of what they want done. Loyalty becomes a *sine qua non* of working with the leader and participating in policy making. And such leaders often shuffle their advisers around, making sure that none of them acquires a large enough power base to challenge the leader’s authority. To some extent distrust of others may grow out of a zero-sum view of the world – when someone wins, someone else loses. The desire not to lose makes the leader question and assess others’ motives. Leaders who distrust others tend to be hypersensitive to criticism – often seeing criticism where others would not – and they are vigilant, always on the lookout for a challenge to their authority or self. Some wariness of others’ motives may be an occupational hazard of political leaders. But leaders low in distrust of others tend to put it into perspective. Trust and distrust are more likely to be based on past experience with the people involved and on the nature of the current situation. A person is distrusted based on more realistic cues and not in a blanket fashion. So we say that *distrust* of others involves a general feeling of doubt, uneasiness, misgiving, and wariness about others – an inclination to suspect their motives and actions.

Although the conceptual discussion of distrust above is quite rich, the coding instructions are rather limited. Beyond a list of 10 nouns to always *code* (in other words, each time one of these nouns appears, that instance should be recorded), directions to code references to specific persons, an admonishment to only code pronouns when the antecedent was unambiguous, and five concrete examples, the following paragraph comprised the entirety of the rule set for coding this sentiment.

“Distrust should be found in text by identifying nouns and noun phrases referring to persons other than the author and to groups other than those with whom the author identifies. Does the author distrust, doubt, have misgivings about, feel uneasy about, or feel wary about what these persons or groups are doing? Does the author show concern about what these persons or groups are doing and perceive such actions to be harmful, wrong, or detrimental to him/herself, an ally or friend, or a cause important to the author? If either of these conditions is present, the noun or noun phrase is coded as indicating distrust. An author’s score on this trait is the percentage of times in an interview response that he or she exhibits distrust toward other groups or persons; the overall score is the average of these percentages across the interview responses being studied.”

Such levels of abstraction and reliance on human competence are typical of the instructions for hand-coded content analytic schemes in the social sciences. Unfortunately they provide very little guidance for the construction of an automated scheme, and this is compounded by the absence of a large corpus of reliable hand coded text against which to test it. The challenge for SSA’s scheme developers therefore was to develop rules that the originators of the distrust measure could understand and approve, and that produced results congruent to those of trained hand coders. To add to the fun, both the scheme originator and hand-coders could be inconsistent, and had to be coaxed to articulate how a particular construction could be further clarified, without resorting to personal perceptions of the relationship between the author and the actor referred to in the text.

[4]Much of this discussion borrows from *Assessing Leadership Style: a Trait Analysis* by Margaret G. Hermann `http://socialscience.net/docs/LTA.pdf`

1.3 The Profiler Port

SSA needed “ordinary” people to be able to use Profiler Plus and they needed to build it quickly. Despite their positive experience with lisp on prior projects and admiration for the language, it was not initially selected for the implementation because of a perceived difficulty of creating the required GUI in either Franz Allegro Lisp or LispWorks. SSA selected Visual Basic 6 to take advantage of its GUI components and MS Access database support and they sacrificed expected performance for the ability of a “hobbyist” coder to put together quickly an engine and GUI that worked, even if subsequently it meant regularly going home for the weekend while Profiler Plus slowly chewed through its texts.

Although performance was always an issue, by Version 4 (October 2003), other defects were driving SSA to look at out-sourcing a rewrite of Profiler Plus. An initial review listed several problems:

- it did not operate correctly in managed multi-user Windows XP environments, for instance storing each user’s files, preferences etc. in the appropriate places;
- on long data runs it issued “low system memory” messages: Profiler Plus 4 (or Microsoft’s “Jet” database engine) had a memory leak of some sort;
- it had a tendency to create output databases larger than 2GB, thereby causing further problems for Jet;
- there were no error handlers: a broken rule could abort a whole data run;
- it couldn’t run more than one coding scheme sequentially on the same set of documents;
- it couldn’t combine rules from different coding schemes, and so abetted large-scale duplication between schemes and high maintainance costs;
- it wrote results into the same Access database in which the coding schemes were stored;
- it did not run on Linux or OS X.

Based on SSA’s successful work with Ravenbrook on other projects, and prior experience with content analysis using lisp language features for pattern matching and map/reduce operations, SSA asked Ravenbrook to take on the port and specified lisp as the development language. The project was commissioned at a meeting the day after the New York International Lisp Conference in October 2003; a prototype was ready five weeks later and a full GUI version the following March; by May 2004 the third-party Java “part of speech” tagger had been abandoned for a native one.

Version 0.1 (in other words: the act of naïvely rewriting the code in Common Lisp) was ten times faster than the VB implementation, and Profiler has continued to eke out performance improvements ever since.[5] When it comes to quantifying this, we note that performance varies considerably with the quality and size of the coding scheme, and that schemes have evolved and expanded in parallel to Profiler; in short it's not straightforward to produce performance figures. However as a rough guide to processing speeds: we used the "Leadership Trait Analysis" scheme discussed above to process 36 documents on a Win7 64bit i5 3.1GHz with 3.9GB usable RAM. 3487 sentences were processed in 42.4 seconds: 12.15ms per sentence.

Profiler Plus is now a mature closed-source application whose source comprises about 30k lines of Common Lisp and LispWorks extensions (primarily to drive the GUI, via its "CAPI" cross-platform toolkit). It no longer degrades over long runs or with many rules loaded; it no longer uses Access to store either its data or its results; it never crashes; it copes well with poor data; it can be driven either from its GUI, or from batch control files specified on the command line, or over a socket interface; its specification has been enlarged over 30 different product versions; and early in 2014 it was ported with minimal effort to both Linux and OS X.

2. THE LANGUAGE OF SCHEMES

2.1 The Coding Engine

A *coding run* in Profiler Plus typically involves applying one or more schemes to several text documents. Both the loading of schemes and the parsing of raw text (for word and sentence boundaries, punctuation, and the like) are sufficiently costly that it's worth caching the results. Beyond that, the main point of interest is Profiler's *coding engine*: a device for applying a single scheme to one document.

The document is represented by a sequence of sentences, and for the most part these are processed in isolation.[6] Each sentence is processed into a series of doubly-linked CLOS instances, each corresponding to one *token* in the sentence: usually a word[7] or punctuation feature. Tokens have over twenty slots which are directly visible to the coding scheme. These slots include the word's original (that is, unmodified) text, its current text, its part-of-speech and – stored in the prosaically named `slot1`, ..., `slot13` – a bunch of temporary values.

Essentially, the scheme interrogates slot values and then either modifies them, or rearranges the sentence, or outputs *results*. The scheme consists of several[8] *tables* along with an XML control file which specifies an order in which to apply them and settings for run-time configurations. The scheme can also:

- *include* other schemes for recursive processing (for instance, this feature is regularly employed to invoke the part-of-speech tagger near the beginning of a more complex scheme); and
- use named *variants*, which selectively comment various tables in or out using a lisp-like `#+` / `#-` syntax.

The tables represent stages in the coding scheme's work. The coding engine applies each table, in order, to every sentence of the document.

Unlike the majority of "natural language processing" software, the approach of Profiler is rule-based rather than Bayesian; the rules use forward chaining. A typical table might consist of anything from one to over a thousand rules. Each rule has an *anchor*, a *pattern* made up of one or more *predicates*, and a *reduction* comprising one or more *actions*; the rule won't be activated on a token unless one of its slots exactly matches (`unicode-string-equal`) the anchor; the predicates are then tested and if they return "true" then the actions are applied.

Here's a simplified example:

```
Anchor:     well
Pattern:    (token: 0 text: well)
            (token: 1 text: run)
Reduction:  (token: 0 pos= adverb)
```

The interpretation of this rule is: if any of the slots of this token are (`unicode-string-equal` to) the string `"well"` we can try the predicate; if more specifically the `text` slots of the current token and the one following it have the string values `"well"` and `"run"` respectively, then perform the following action: set the `pos` slot of the current token to the string `"adverb"`.

Internally each table is stored as a `unicode-string-equal` hash-table which maps anchors to lists of rules; the coding engine loops across each token's slots looking for matches. Occasionally, rules are written whose patterns are designed to be tested against every token in the sentence; in effect a "universal anchor" (we use the string `"%every%"`) is required.

```
Anchor:     %every%
Pattern:    (not token: 0 slot10: %null%)
Reduction:  (no-repeat (clear 0 slot10))
```

An elegant and efficient solution to this requirement is to create a special hash-table in which lookups necessarily always succeed:

```
(make-hash-table :test 'true
                 :hash-function (constantly 0))
```

and insert into this one entry – the choice of key doesn't matter! – mapping to the rules in question.

In theory we keep matching a table against each sentence until all possible patterns have been matched and acted upon; in practice, managing this requires a little subtlety. If a pattern returns "false", or if the reduction doesn't modify anything – because all it does is generate output – then processing of the current table continues from the next token in the sentence. However if anything has changed (e.g. a slot value has been rewritten) then the engine *backs up*

[5] With the release of LispWorks 5 in July 2006, LispWorks for Windows (and Linux) inherited the architecture of the other 32-bit LispWorks implementations, delivering 30-bit fixnums (up from 24) and a much improved garbage collector; processing speeds improved by about 20% overnight.

[6] It is however sometimes necessary to peek back from one sentence to its predecessors. *Pronoun disambiguation* – making sense for instance of `"Marc has arranged a great conference. He worked very hard."` – is a typical reason for doing so.

[7] Because languages like traditional Chinese are written without spaces between words, in some coding schemes the tokens will initially represent single *characters*.

[8] Three dozen in the English part-of-speech tagger, for example.

two tokens before resuming its work; this is done to catch any patterns which – as a result of the change – will now match on neighboring tokens. Unfortunately this practice carries with it the risk of *looping*, sometimes by a single pattern, in more complex cases involving the interaction of several. (For example, consider three rules, the first of which matches `"alpha"` in `slot1` and converts it to `"beta"`, the second rewrites `"beta"` as `"gamma"`, and the third `"gamma"` as `"alpha"`; and none of these rules perform any secondary tests.)

If looping is detected (50 changes, say, without making any headway through the sentence), then we assume that no further progress can ever be made and we move forward to the next token regardless. When we get to the end of the sentence, if changes have occurred anywhere along the way, then processing reverts to the beginning of the sentence. This is done to catch patterns which now match as a result of "non-local" changes, and is another potential cause of looping. So each time the sentence is repeated, the looping sensitivity is halved and when this value gets too small Profiler issues a `"Non-local loop"` warning, and we take the current table on to the next sentence. Looping is sometimes a useful construct and sometimes less so; originally scheme developers controlled it by writing increasingly subtle predicates to spot it, but in more recent years a `no-repeat` action has been introduced to make the process much more straightfoward. Unfortunately `no-repeat` encouraged lazy rule construction and, along with the `"%every%"` anchor, is now mandated to be used only when absolutely necessary.

The source file for each table is XML (which works well with the external tool used for editing the rules), but as shown above the predicates and actions look a lot like S-expressions. Indeed, the lisp reader is used to parse them, albeit with a somewhat modified readtable. The trailing colons (for instance, in `text: here`) and equals signs are optional and purely stylistic: the readtable uses them as delimiters but otherwise ignores them.

2.2 Predicates and Actions

We've met the predicate called `token` and an action with, as it happens, the same name. There are about twenty different predicates, and over thirty actions; we give a few further examples below. Essentially, these operators form an embedded programming language which has evolved gradually over the last decade and whose task, as already suggested, is to take tokens as its input, selectively update these tokens, and occasionally output a result.

The syntax of the `token` predicate is

```
(token offset [key value]*)
```

If `offset` is a number, then `0` means the current token (that is, the token which matched the anchor), `+1` is the following token, `-1` the previous one, and so on. The `key`, `value` pairs name slots and give values to be matched; however the special key `newlabel` is used to associate a symbol with the current token, so that later predicates and actions in this rule can refer to it by name (for example, as the `offset` of another `token` predicate).

`Token` is by far the most used predicate. As a result it has attracted both the most optimisation effort and also the most extensions. Three examples:

- the simple use of wildcards

```
(token 3 lemma (any-value band generation
                          style cloth*))
```

- instead of a fixed string value, a restricted `format` operator can be used to generate match strings on the fly, using slot values from the current token and its neighbors:

```
(token: 0 slot1 (format "(both ~a)"
                        (-1 slot10)))
```

- instead of a string value, the `any-value` operator can be used to offer a choice of values, and within that the `file` operator specifies that these values will be found in a separate data file:

```
(token 0 text (any-value
               (file "locations.txt")))
```

  Similarly, the `any-slot` operator allows a match to occur in a range of slots.

Sometimes, asking whether some slot/value combination is to be found some exact distance away from the current token is too specific: natural language is more fluid than that. So within the scope `variable` operator, offsets are interpreted as "no more than". In the following example, the `lemma` slot of either this token or its immediate predecessor must be one of `"north"` or `"south"`:

```
(variable
  (token: 0 pos: punc lemma hyphen)
  (token: -1 lemma (any-value north south))
  (token: 1 lemma (any-value east west)
         newlabel: c))
```

By default predicates are combined as if by `and`: all the predicates in a rule must succeed. This can be overridden with a combination of `or`, `and` and `not`[9] predicates, each of which takes further predicates as its arguments.

We have already met the action `token` which sets slot values. Other actions which modify the sentence include: `insert` which creates new tokens; `delete` which removes them; `split` which divides one token into several smaller ones, breaking on a given character.

By default actions are combined as if by `progn`: if the predicate has succeeded then all the actions are performed, in order. This behavior can be modified, with operators such as `when`, `unless`, `if` and `progn`:

```
(when (and (not token: -1 pos: be)
           (not token: -2 pos: be))
  (insert destination: 0 before: yes
         lemma: be pos: be tense: present))
```

In theory, every rule's pattern could be left empty, and its predicates moved into a `when` operator encompassing the actions. However, SSA prefers to keep the predicates logically separate, to allow use of priority values to control rule execution order and to promote readability for scheme developers. In addition, although `when`, `unless`, `if`, and `progn` all have necessary uses, they feed the tendency of some "schemers"

[9]There is another variety of `not`, which behaves as a `token` test whose result is then negated.

to write fewer, complex rules rather than more, simple rules. SSA prefers more but simpler rules for ease of maintenance.

Among the actions which generate output are `csv` and `xml`. These, along with the `token` and `insert` actions, have access to the `format` operator mentioned above. In these contexts (but particularly for output) the strings which `format` builds might also incorporate values from the current *environment*, such as the date on which the source text was generated, the word count or full text of the current sentence, or even the pattern being matched.

If the work of a predicate or action is straightforward, then so by and large is its definition. For example, in the best spirit of lisp interpreters, the Profiler sources read:

```
(defpredicate or (token match)
  (dolist (submatch match)
    (when (apply-predicate token submatch)
      (return-from or
        t)))
  nil)


(defaction when (token match)
  (let ((condition (first match))
        (rest (rest match)))
    (when (apply-predicate token condition)
      (execute-progn token rest))))
```

The defining macros `defpredicate` and `defaction` are a thin gloss around `eql` methods on the generic functions `execute-predicate` and `execute-action`; their main purpose is to assist the LispWorks editor to locate the forms when given a predicate's or action's name (so the developer can look for the definition of `when`, for example).

Finally, here's a real rule, lifted directly from one of the coding schemes. As a piece of computer code, it's neither beautiful nor particularly high-level; but the abstractions it uses are right for the people who have to maintain it.

```
<Rule Anchor="conj" PatternNumber="76">
    <Pattern>
(variable
 (token: 0 pos= noun slot1= dt)
 (token: 1 pos= verb newlabel= v)
 (not token: 1 lemma (any-value arrive snore sneeze
                                sit die come lie))
 (token: 2 pos= conj newlabel= a)
 (not token: a conjunction: or
      slot9= clause slot9= sentence)
 (token: 3 pos= verb newlabel= b)
 (variable-offset from: b distance: 100
                  pos= punc newlabel= p)
 (not-any start: b end: p pos= conj))
    </Pattern>
    <Reduction>
(copy start: 0 destination: a)
(set s-token: 1 s-slot: modifier
     d-token: b d-slot: modifier)
(set s-token: 1 s-slot: truthvalue
     d-token: b d-slot: truthvalue)
(token: a slot9= clause)
(copy start: b end: p destination: v exclude= yes)
    </Reduction>
</Rule>
```

If this rule is applied to `"The man laughed and cried after the maid sang."`, it will convert the sentence into `"The man laughed after the maid sang and the man cried after the maid sang."`. The `"and"` will be marked as a “clause boundary”: a complete clause now exists on either side.

2.3 Implementation

Profiler's rule language is run interpreted. A typical large scheme might consist of several thousand rules, many of which aren't touched during any given coding run. Just reading and parsing all these rules takes a significant time. Further, considering even those rules whose predicate does get invoked, most predicates fail and so most actions are never fired. Finally, most predicates in a scheme will be calls to `token`, often in its most straightforward form. Bearing these observations and the results of some simple code profiling in mind:

- scheme source files (XML) are parsed and then stored back to disk in a simple binary format which in particular precedes each rule's strings (anchor, predicate, action) by their lengths – this makes subsequent loading of the scheme significantly faster;
- predicates and actions are held in memory as unparsed strings until they are invoked, whereupon they are parsed into and retained as S-expressions;
- some predicates are then rewritten, in terms of simpler operators which will run faster;
- most effort went into optimising `token`; and
- in very simple cases, `token` can be replaced by a compiled closure:

```
(defun optimized-token (token slots-and-values)
  (loop for (slot . value) in slots-and-values
        always
        (unicode-string-equal (funcall slot
                                       token)
                              value)))
```

Note that for each slot visible to the coding scheme, there is a reader with the same name. Invoking (with suitable guards around the `funcall`) standard readers is noticeably faster than calls to `slot-value`.

```
(defclass token ()
  ((text :accessor token-text :reader text)
   ...))
```

2.4 GeoNames

Profiler has the ability to specify a number of “special tables” in a scheme's configuration. These are denoted by keywords and include `:hold` which means “don't process any sentence beyond this point until you've processed all sentences before this point” and is useful for pronoun and entity disambiguation; `:split-words` which splits every token into its constituent characters; and `:geonames`.

GeoNames[10] is a free-of-charge database which contains details of over eight million placenames, in the form of tab-separated strings. It's sometimes useful for a scheme to know whether it's potentially looking at a placename and if so to access other details about the place, such as which

[10] `http://www.geonames.org/`

country it might be in; the `:geonames` special table walks the sentence looking for potential matches. We don't want to encounter any noticeable delays while doing this lookup; and we'd rather not get tied back into SQL as a means of performing it. So GeoNames data files are walked once (the first time they're loaded) and each converted into two binary files: a dictionary and an index.

The dictionary is large and is never loaded in its entirety; the index is small and can be reloaded cheaply in future invocations of Profiler. Doing so creates a hash-table whose keys are the strings which start GeoName phrases, such as `"South"` or `"Cambridge"`. Values in this table are either a number (an index which can be handed to `file-position` for random access into the dictionary file), or a data structure if this string has been visited before. Some complexity arises because a phrase such as `"Cambridge"` might be a complete placename, or the beginning of over a hundred of longer phrases, such as the `"Cambridge Community Resources Building"` (in Nevada). Profiler returns the longest possible matches – and there might be several – and inserts them into the sentence for further consideration.

Any scheme which incorporates `:geonames` must beware of false positives! For example, while procesing the phrase `"Vladimir Putin phoned President Barack Obama on Friday"`, we learn that `"President"` is the name of several hotels (there's one in Andorra); there are nine places called `"Obama"` in Japan, and there's a `"Friday"` somewhere in Texas. In general, most short English words turn out to be a placename somewhere or the other. For this reason, two control files were added to `:geonames` processing: `"stop-words.txt"` and `"stop-pattern.txt"`. Any token found in `stop-words` is ignored by `:geonames`, along with those that fail the test in `stop-pattern` which currently reads:

```
(or
 (token: 0 slot8 (any-value international-region
                            country))
 (not (token: 0 slot11: GeoName-candidate)))
```

2.5 The Lisp

Down at the language level, two aspects of the implementation are worth mentioning. The first is string processing, and the morals for the modern programmer are simple: don't even think about using 8-bit strings; and don't even think about string I/O that isn't UTF-8 safe. Below the hood there was plenty of work to do.

- The implementation shadows `cl:make-string` and `cl:with-output-to-string`, issuing compile-time warnings unless either `:element-type` or `:believed-safe` is set.

- It shadows `cl:with-open-file` with a macro which takes two additional keywords: `:skip-bom` for controlling the handling of byte-order marks, and `:explain` for specifying what should happen when an input file fails to conform to its expected external-format (UTF-8, typically). The LW extension `hcl:file-string` is also shadowed, by a function which understands `:skip-bom`.

- It hacks into LispWorks' file-encoding algorithm, to allow a run-time choice of default encoding between `:latin-1` and `'(win32:code-page :id 1252)`. This feature made a lot of sense back in 2004 but it doesn't any longer: although this hack was never taken out the only format supported today is UTF-8. If source documents don't conform to that then Profiler provides the tools to convert them.

Then we come to Profiler's heavy use of macros and closures. It's hard to come up with non-trivial but succint examples, so we satisfy ourselves with brief descriptions of a few of the macros which Profiler uses.

`with-data-file` is a wrapper around `with-open-file` for reading text data files: its body is a loop for processing one line of text. Parameters include control over whitespace stripping, forms to run even when a line is `#`-commented out, and the name of an explicit `block` (for moving onto the next line):

```
(with-data-file (line ifile)
    (:block continue :strip t
     :always ((incf count)))
  ...)
```

`(with-progress ((funvar) &body body)` binds `funvar` to a closure which controls a progress bar and the text above it, in the Profiler GUI. This closure can be passed a number (to move the bar), a string, or a keyword such as `:halt` to control both the coding engine and other aspects of the GUI. This macro / closure combination appears 75 times in Profiler source code – a powerful tool which has required zero maintenance in over ten years.

`with-suspended-richedit-undo` appears just once. Something had to be done to prevent the Windows rich-text widget from building a character-by-character undo list out of syntax-coloring changes. The ugliness (involving, unfortunately, 100 lines of C++) is elsewhere.

`writing-with-caution` is a wrapper around `with-open-file` with implicit `:direction :output`. It ensures cleanup (logging, and removal of potentially corrupt or part-written files) if any error occurs within its scope.

2.6 Parallelism

By 2006, Profiler Plus was in use for assessment-at-distance in a number of government and research settings, and the performance gains provided by the port to lisp were more than adequate for those tasks. However, new large scale projects loomed on the horizon, including:

- a project that would require repeated coding of a 10 year corpus of news stories,

- a dynamic web-based tool for media monitoring to provide analysts with near real-time access to data from millions of Arabic language media documents directly from the source without the need for, nor the inherent risk of, intermediary processes, screenings, and translations, and

- national brand monitoring[11] which would require daily processing of up to 25 million news articles in several languages.

[11] `http://www.eastwestcoms.com/global.htm`

To meet the scale of the processing challenges in these production settings, SSA considered parallelism on multi-core machines. Various options were considered:

- running different schemes in parallel (this comes for free with Profiler's socket interface, but we still need some form of controller to spot which threads have completed and are ready for new tasks), or
- different documents within a given scheme, or
- different sentences within a given document (pronoun disambiguation notwithstanding).

However, given the availability of very cheap single core Windows boxes, SSA decided to simply add boxes to a bank of dedicated hardware and use an external controller to manage many single-core instances of Profiler Plus running batch jobs via its socket interface. SSA wrote the controller in Visual Basic – a tool with which they had experience – leaving Ravenbrook to focus on the core engine and GUI in lisp. After the recent port to Linux, cloud-based scaling has become an attractive and cost-effective alternative to maintaining racks of individual machines.

3. PROGRAMMER AIDS

3.1 For the Lisp Developer

As the application has evolved over the years, so have the debugging tools used to fix and extend it. One of the most useful of these has been present since the very beginning: the global environment.

The coding engine is controlled by an instance of the class `global-environment`: this object points to the sentences which comprise the document, the coding scheme's tables and their running order, configuration values, dictionaries, the GeoNames index, results, logs, the progress function, and 20 other global values pertinent to the run. The variable `*global-environment*` points to this object; at run time the *only* use of this variable is to scavenge dictionaries and schemes from previous runs when constructing the next global-environment. For all other use, the global-environment is passed around as a function parameter (and also, for code clarity, often returned as a function's result); also tokens point back to the global-environment. In short, the application routinely accesses this object without reference to a special variable.

However as a debugging aid and in combination with standard lisp development tools – in particular the debugger, the inspector and the inestimably useful code stepper – `*global-environment*` is indispensable: it gives the developer immediate access to every aspect of the coding process.

Another feature present since the beginning is the macro `handling-unexpected-errors`, which is wrapped around every execution and GUI thread. This establishes a `handler-bind` for `warning` and `serious-condition`, and ensures: appropriate `abort` and if possible `continue` restarts; context-specific logging with full backtraces as appropriate, via CL-LOG[12], to file and (with less detail) to a logging window; and a fault recovery strategy which is configurable for different users:

- During a production run, the best thing to do with errors (up to some configurable maximum) is to log them and continue the run: if a later analysis reveals significant problems with the coding scheme then the run can be repeated.
- When schemers hit an error, either they can tell straight away what the problem was or they're going to have to ask someone else for help: either way their best strategy is to halt the run.
- No end user should ever encounter the lisp debugger (let alone the Windows console, or worse still a vanished application); but when lisp developers hit an error, they want to be offered the debugger.

It's worth mentioning at this point that there are two different "builds" for Profiler Plus: one for the developer which offers the full LispWorks IDE, whereas that for schemers and end users is a royalty-free application from which the IDE is absent. The two executeables are essentially identical: apart from the obvious, the only real difference is that the development version performs a `load-system` at startup, to ensure that its sources are up-to-date.

Both builds load precompiled *patches*[13] at startup. The process of interpreting a user's backtrace, reproducing the problem locally and then generating a patch is lightweight, often taking little more than an hour from beginning to end, and has proved a highly effective means of maintaining the application in between product releases.

3.2 For the Coding Scheme Developer

It took some time for either party to realise that the coding schemes were computer programs and therefore the schemers were computer programmers, to whom all the issues surrounding software development applied. One consequence of this realisation was the introduction of more careful curation for the different versions of each scheme; and out of this came a number of features in Profiler to support versioning in schemes. In particular it's worth mentioning *compressed schemes*: single files (`.zip` under the hood, and managed by the CL-ZIP library[14]) distinguished by a version number as well as the scheme name.

Profiler Plus also supports a number of debugging aids aimed specifically at the schemer. Logging is the first line of defense. In addition to the configurable logging levels provided by CL-LOG,

- there's a setting which (in return for slowing coding runs down by about 20%) automatically generates a report showing which rules were responsible for loops, warnings and errors, and annotates these conditions in the application's log window with additional detail to make debugging them easier;
- another setting, in exchange for a 5% performance hit, generates statistics files showing which rules did or did not fire;
- within the scope of the `debug` predicate, whose body is an implicit `and`, (and provided yet another runtime

[12]`http://www.nicklevine.org/cl-log/`

[13]As described in `http://www.nicklevine.org/play/patching-made-easy.html`. Throughout Profiler's 86 releases to date, some 250 patches have been issued.

[14]`http://common-lisp.net/project/zip/`

setting is enabled) all predicates log their progress in greater detail;

Here's a short sample extract from an "execution statistics" report. Each rule is identified by a "PatternNumber", and the first few characters of its pattern and reduction.

```
GB01PreliminaryCleanup
======================

None of the patterns in this table fired

GB02BadLexicon
==============

The following 1 pattern fired 2 times

5577  capitulatio...   (token: 0 lemma: capitu...
                       (no-repeat)(token: 0 sl...

The following 16 patterns fired 1 time

3252  regret           (token: 0 lemma: regret...
                       (no-repeat)(token: 0 sl...
3873  tension          (token: 0 lemma: tensio...
                       (no-repeat)(token: 0 sl...
814   crisis           (token: 0 lemma: crisis...
                       (no-repeat)(token: 0 sl...
(etc)
```

The introduction of "condition summary" and "execution statistics" reports has eliminated the need to post-process the less specialised logs, and so saved time and effort for the schemers.

Two other invaluable tools for the schemer are the tracer and the stepper. Both are configured on a per-table basis. The tracer writes a log entry – in much greater detail than the "statistics" summary – each time a rule fires. It generates a lot of output; here's a short log extract. Each rule is described by its "PatternNumber" and the names of the actions it invoked; and then the token which matched the rule's anchor is identified. Note how table `DenominalVerbs` is repeated over the whole sentence, as described in the discussion of looping above.

```
Reducing sentence #5 with table DenominalVerbs
DenominalVerbs 4 when,when,when,token Token #15
DenominalVerbs 4 when,when,when,token Token #23
Reductions limit reduced
Reducing sentence #5 with table DenominalVerbs
Reducing sentence #5 with included table GoodBad
Reducing sentence #5 with table PreliminaryCleanup
PreliminaryCleanup 24 no-repeat Token #4
PreliminaryCleanup 28 no-repeat,token Token #4
...
```

The Stepping Tool (Fig. 1) allows the the scheme developer to halt the computation at desired points, examine the values in each slot, and – using the "Back" button – flick between two stages to get a quick visual comparison. (This tool is also useful for lisp developers, as it gives control over when to set breakpoints in often-invoked code and start stepping through the lisp itself.)

There are about 30 settings which control how the coding engine works: not just the error handling policy and control over logging already mentioned but: choice of language model, tokenisation policy, the Treatment Of Successively Capitalised Words, interpretation of ambiguous date formats, looping detection sensitivity, and so on. Settings can be changed via an Options Tool in the Profiler Plus GUI, in batch control files:

```
<ppb>
   <schemes>
     job003718/master,,
   </schemes>
   <documents>
     /home/nick/Profiler/docs/test.csv
   </documents>
   <columns>
     <column type="pass-through">userid
     </column>
     <column type="coding">comments
     </column>
   </columns>
   <options>
     :execution-statistics t
     :rules-case-fold t
     :unknown-tag-error :pass
   </options>
 </ppb>
```

or by an `<options>` tag in an individual scheme's XML control file.

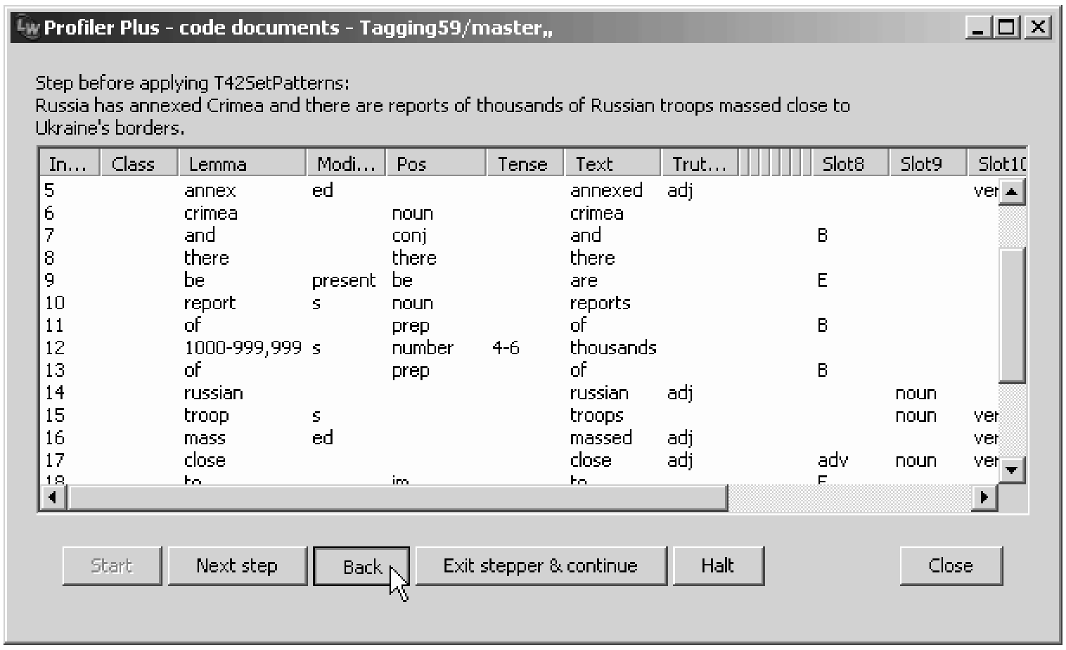
In addition to the log window, stepping tool and options dialog already mentioned, the Profiler GUI provides a "Code Documents" pane which allows the schemer to select schemes and documents; when they activate a run each scheme will be applied to every document. Schemers can mark up, view (with syntax-coloring for the markup), convert into UTF-8 and validate their documents; they can view and modify the overall structure of coding schemes; they can generate and test out simple batch files. Only the editing of rules is elsewhere; it's surprising that it took so long for anyone to identify this GUI as a programming environment.

4. ALTERNATIVE APPROACHES

SSA chose to approach the problem of slow and unreliable human content analysis by implementing a rule-based general-purpose text coding engine and using it for fully-automated text-coding. In retrospect, was this a good choice? We consider here some alternative methods:

- continuing with hand-coding;
- continuing with hand-coding, but distributing the workload[15];
- computer assisted coding using something like *QDA Miner*[16];
- computer assisted coding with human verification[17];
- using a Bayesian/machine learning approach to automated coding.

[15]`http://www.umass.edu/qdap/index.html`
[16]`http://provalisresearch.com/products/qualitative-data-analysis-software/freeware/`
[17]`http://www.umass.edu/qdap/IJMRA.pdf`



Step before applying T42SetPatterns:
Russia has annexed Crimea and there are reports of thousands of Russian troops massed close to Ukraine's borders.

In...	Class	Lemma	Modi...	Pos	Tense	Text	Trut...	Slot8	Slot9	Slot1
5		annex	ed			annexed	adj			ver
6		crimea		noun		crimea				
7		and		conj		and		B		
8		there		there		there				
9		be	present	be		are		E		
10		report	s	noun		reports				
11		of		prep		of		B		
12		1000-999,999	s	number	4-6	thousands				
13		of		prep		of		B		
14		russian				russian	adj		noun	
15		troop	s			troops			noun	ver
16		mass	ed			massed	adj			ver
17		close				close	adj	adv	noun	ver
18		to		im		to		E		

Figure 1: The Stepping tool, showing a sentence part-way through its part-of-speech analysis.

Hand coding, computer assisted, distributed or otherwise, with or without verification. There are services which offer expert human coding and evaluation of coding, such as the the *Qualitative Data Analysis Program* at the University of Amherst. SSA uses similar processes to develop test suites that are used to gauge the accuracy of the automated coding schemes. However, many of SSA's most important projects require near real-time analysis of large volumes of continually updated text feeds. Scaling human coding efforts to achieve this requirement is simply cost prohibitive. In addition, the scope of analysis available per unit of cost is much greater with automated coding. For example, in 1983, the quite respectable volume of text required for an at-a-distance assessment of an individual was 50 100-word text samples (5,000 words total) and the limiting factor in the process was the time and cost for each of the seven traits to be coded. Today researchers using Profiler Plus routinely examine hundreds of thousands of words spoken or written by a single assessment subject; the availability of documents has become the limiting factor.

Bayesian machine learning automated coding approaches. For better or worse, the original designer of Profiler Plus was reasonably well trained in pattern-matching and averse to statistical approaches to text coding that start by discarding "information that will likely be unhelpful, ancillary, or too complex for use in a statistical model."[18] These approaches may be natural to anyone who views text as "unstructured" data or who is trained in statistical approaches to data analysis. However, in statistical approaches a highly structured, albeit complex, form of symbolic communication (text) is de-structured to make it amenable to statistical analysis and much is lost in this process. Rule-based approaches also offered a more accessible and transparent coding process in which all the steps leading to any particular coding can be examined. As Profiler Plus was intended to be a platform used by the "average" social scientist for the development of coding schemes, this transparency was crucial. Last but not least, no fully annotated corpuses existed for any of the target assessment coding schemes. When hand-coded texts were available, the annotation often consisted of notes in the margin indicating one or more codes that applied to the paragraph (or in the best cases to a single sentence). It may be the case that a statistical approach would perform just as well on any of the text coding tasks Profiler Plus takes on. However, SSA has not yet run up against any limitations of Profiler Plus – or the coding scheme language – that Ravenbrook has been unable to remedy.

One external measure of the appropriateness of the approach is the adoption of Profiler Plus in the social science research community. In the mid 1990s, perhaps half a dozen researchers used "Leadership Trait Analysis" (LTA). Today more than 300 researchers are using LTA on Profiler Plus through a program administered by Syracuse University and, in the last year, 166 researchers registered to use one or more of the wider range of coding schemes now available on `profilerplus.org`.

Profiler Plus running a well-constructed coding scheme is not as good as a human expert, but is more reliable and much, much faster. Profiler Plus running a well-constructed coding scheme is better than the average well-trained human coder. Profiler Plus does not learn, but it can be taught.

5. COMMERCIAL APPLICATION – THREAT TRIAGE

In 1999, one of the FBI's problems was the large number of disturbing or threatening communications they were asked to analyze and assess for risk. Unfortunately they

[18]Grimmer, J., & Stewart, B. M. (2013). *Text as data: The promise and pitfalls of automatic content analysis methods for political texts.* Political Analysis, 21(3), 267-297.

lacked a reliable method for distinguishing urgent cases of imminent harm from the much more numerous cases where the threatener was simply blowing off steam or the communication was just a malicious hoax. At the request of then Supervisory Special Agent Sharon Smith at the FBI's elite Behavioral Science Unit, SSA provided Profiler Plus and a number of coding schemes to assist her in evaluating the numerous competing and often contradictory methods used for the assessment of threatening communications. By 2006, the now Dr. Smith's research had yielded a practical result[19]. She developed a threat assessment method that used a largely hand-coded content analysis system to classify threatening communications into 3 groups: low risk (in 90% of cases no further action occurs), moderate risk (in 30% of cases the threatener takes further action), and high risk (in 67% of the cases the threatener takes further action, up to and including murder and sexual assault). However, although a vast improvement on previous methods of assessment, this process remained labor intensive and not widely available to the thousands of security and law enforcement agencies in the United States; and Dr. Smith had no wish to be on call 24/7/365 to conduct assessments. In 2011, SSA proposed that they automate the 5 coding schemes used by Dr. Smith that did not already run on Profiler Plus in order to provide threat assessment via a web site available to users around the country, day or night. Thus `ThreatTriage.com` was born and today users from Federal, State, and Local governments along with military and civilian police departments and private corporations receive threat assessments within seconds of submitting a disturbing or threatening communication to the threat triage web site. The only change to Profiler Plus required to meet this challenge was an extension of the socket interface to allow texts and results to be exchanged directly between Profiler Plus and the controller rather than being written to disk for periodic collection.

6. FINAL REMARKS

What Profiler Plus does is quite sophisticated and most of the sophistication is in its coding schemes. In social science it has raised the bar for the acceptable minimum amount of text that can be used to draw inferences about individuals from 5,000 words to over 100,000 words; it has enabled social scientists to extend the scope of their inquiries in time and across individuals. As a Common Lisp success story whose telling has been long overdue, we can't claim that the project would have failed if it had been implemented in any other language, only that for all manner of reasons implementing it in lisp made the work much more straightforward. And considerably more fun.

[19]`http://forensicpsycholinguistics.com/fp/docs/Sharon_Smith_Dissertation.pdf`