Todo: <note> <A comment> (the idea of putting a special comment on the present context, which is carried over to the output.)

STICK-E NOTES: the Context Matcher User Manual

Peter Brown
February 21, 2001

ABSTRACT

This document explains the components of the current implementation of the Context Matcher and gives full details of the syntax used.

Introduction

The purpose of the Context Matcher is to be a general engine that will can act as the core of a wide range of context-aware applications. The Context Matcher uses *stick-e* notes, which are the electronic equivalent of Post-it notes. A stick-e note consists of some information coupled with an electronic context. The Context Matcher (we will just call it the "Matcher" henceforth) has two key sets of data:

- a set of stick-e notes representing a document collection. In principle, these may (a) reside throughout in a database (the current set of notes may only be a small part of the whole database) or (b) they may be downloaded at the start of a session from (b1) files or from (b2) a database. In the current implementation (b1) applies, and the loaded notes are treated as static, i.e. they are loaded from a file at the start of a Matcher session, and if they change the Matcher needs to be re-started (but see the issues for discussion at the end).
- the *current context* (previously called the *present context*), which describes the user's current status, e.g. location, orientation, temperature, companions, perhaps even mood. This is typically set by a combination of automatic sensors and direct actions by the user.

Given such data the Matcher can perform such operations as:

- triggering those of the stick-e notes that match the current context (the proactive case).
- performing a match in the opposite direction, whereby the current context is used as a query to extract information from the stick-e notes (the interactive case).

The original thinking behind the Matcher is described in the paper *A General Mechanism for Context-aware Matching and Conversion*. See http://www.cs.ukc.ac.uk/people/staff /pjb/papers/karlsruhe.ps for more details. In particular the Matcher logically consists of a retrieval activity that finds out which notes best match.

This document explains how to use the current implementation of the Matcher. It has been implemented in Java. We start by explaining the comparison activity. Those readers, however, who learn best by example, might want to skip over the explanations and look ahead to the examples at the end of this document.

THE FIRST STAGE: MATCHING

Representing contexts

A unifying concept in the Matcher is that almost all its data is represented in the form of a context. Each stick-e note in the document collection is represented by a context, and so is the current context. (Although the current context has the same syntactic form as a stick-e note in the document collection, we use the term "stick-e note" exclusively to refer to a document in the collection.) We therefore start by describing how contexts are represented. A context is a set of *fields*, where each field consists of a name and a *value-tuple*, which consists of one or more individual values. Often the value-tuple is just a single value, in which case we just use the term "value". For input/output purposes a context is represented in a simple SGML form called a *scontext*, and has the name of each field enclosed between '<' and '>'. An example of a scontext (taken from the paper cited above) is:

```
<temperature> 23
<facing> 80..100
<with> printer_LP7
<location> 888, 999
```

The above scontext contains a `<temperature>` field, with the value 23, and a `<facing>` field, showing the compass orientation where the user is currently facing. The notation for the value of `<facing>` shows it to be a range: 80 to 100. Thus a range consists of two numbers with `..` separating them. (Currently ranges of strings or other non-numeric values are not supported.) Ranges can be written in either order, e.g. `3..5` is the same range as `5..3`. After the `<facing>` field the scontext contains a `<with>` field saying that the user is near a certain printer, and a two-dimensional `<location>` field. A multi-dimensional value consists of a sequence of single values (which might be ranges), separated by commas. Thus an example of a value-tuple representing a three dimensional field is:

```
<readings> 34..56, true,
   -3..3
```

Ranges can be infinite, e.g.:

- `10..` means any number greater than or equal to 10.
- `.. 6.8` means any number less than or equal to 6.8 (including negative numbers).
- `..` means any number.

The name of a field must be a sequence of letters and/or digits, started by a letter. The characters '_' and '-' are also allowed, apart from as the initial character. Following the tradition of SGML, names are not case-sensitive, though all other components of the Matcher are. There are a few reserved field names — see later; otherwise the Matcher just accepts whatever field names it finds in its data — there is no concept of declaring them. To allow for XML notation a field name can be written < *name* />, though this is not, in fact, correct SGML. Names can be followed, in the normal SGML way, by attributes: values of attributes should be enclosed within double-quotes, e.g. `<N ATT="val">`.

Currently there are just two basic data types: numeric and string. The Matcher can be configured either to use integers or to use real numbers. (In earlier implementations, each name was associated with a unique data type, i.e. if one `<x>` field was a number, they all had to be. However this turned out to be too restrictive and now a name can be associated with any value.) The data type of a field is determined by the appearance of the value: if the first character is a digit (possibly preceded by a minus sign and or a decimal point) then the value is treated as numeric (Todo: this is rather clumsy, but can we live with it? The alternatives may be declarations of data types, etc. Currently everything that is not a number is treated as a string.). If it is required that a value be considered as a string it can be placed in quotes, e.g.

```
<CompanyName> "600 group"
```

SGML comments are allowed between or within fields, e.g.

```
<facing> <!--west--> 270
```

(As a safety measure for unmatched openers for comments, the Matcher gives an error if the body of a comment includes `--` characters.) A comment is taken as part of the value of the field that precedes it, and it is a good convention to write an explanatory *after* rather than *before* the field it relates to. The advantage is that the comment then appears when the value appears, e.g. in error messages or in the final output; an example of a comment is a place-name such as `<!--Blean-->` placed after some co-ordinates representing the location of that place.

As with all SGML-based notations, whitespace is ignored.

Values must not normally include the reserved character '<'. However there may often be a need to provide values that are in HTML, and to cater for this, the Matcher provides the `<markup>` tag. An example of its use is:

```
<body> XXX
  <markup> <tag1> ..
  <tag2> ..</tag1>
  </markup>
  YYY
```

Here everything will be treated as the value of the `<body>` field, since the tags within `markup` are not recognised by the Matcher. `<markup>` tags may not be nested within one another. (PJB comment: markup tag is a low priority.)

Samples

Now deleted.

Metavalues

There are facilities for *metavalues*, which match any value. There are two currently implemented metavalues: `ANY` matches any value-tuple, whereas `any` matches any single value; thus, for example, `any,29,any` matches any three-dimensional value-tuple that has `29` as its middle value. `ANY` must occur on its own as the sole value of the field; otherwise an occurrence of `ANY` is treated as an ordinary string.

Metavalues can also be used as tag names: `<ANY>` matches any other tag name. There is no difference in meaning between `<ANY>` and `<any>`.

Active fields

Being *active* to a stick-e note means taking part in the matching process. The concept applies at two levels. At the higher level a whole stick-e note may be active or not. At the lower level individual field names may be specified as being active: e.g. the name `<facing>` may be specified as an active field name; this means that if a note includes a field of that name then the field will be part of the matching process. (Often only a subset of the fields of a note are active in the matching process.) An individual field is active if its name is active and its lies within an active note.

Normally a note is active if and only if at least one field within it is active.

The purpose behind the idea of active fields is as follows. There may be a host of sensors that are contributing fields towards the current context, and at any one time only a subset of these are likely to be relevant to the user/application. Similarly the user/application may wish to focus triggering on certain active fields of notes ("I am not currently interested in information triggered by the temperature"). Removing a field from the list of active ones can achieve these effects. We explain later how the names of the active fields are specified.

Matching of fields

A core operation of the Matcher is matching individual fields, and deriving an overall score for a match. A field of a stick-e note is only matched against the current context if it is active (or vice-versa if matching is interactive rather than proactive — in the rest of the discussion below we assume the proactive case). Assuming it is, a further prerequisite for two fields to match is that they have the same name (though the name can be lower-case on one side and upper-case on the other) and that the two value-tuples have the same data type. Finally the values are compared to derive a matching-score. All numbers are treated internally as ranges: thus the number `10` is treated as the range `10..10`. With this convention, two numeric values are scored according to how well their ranges overlap. String matching simply involves testing if one string is included in the other, though later this might be made more sophisticated. Two multi-dimensional value-tuples cannot match unless they have the same dimensionality. Overall if a compulsory field does not match or gets a matching-score of 0, then the complete match fails.

Labels on fields

(PJB comment: labels are low priority.) An extra concept that affects matching is *labels* on fields. To understand the purpose of labels, consider the following situation.

There are three temperature sensors in a room, each forming part of the current context. One way of representing them is to use a different tag for each, e.g. `<temperature1>`, `<temperature2>` and `<temperature3>`.

A more flexible approach, which better captures the similarity between sensors, is to use the labelling facility. A label is prefixed to a field value, e.g.

```
<temperature> sensor1= 63
<temperature> sensor2= 64
<temperature> sensor3= 61
```

Almost any string of characters can be used as a label, though obviously a label cannot include an equals sign. The label must occur on the same line as the field name. (If there is a real equals sign in a field value, the value should be made to start on a new line, thus avoiding it being taken as part of a label delimiter.)

The above labelled temperatures give more flexibility in the matching process: essentially a stick-e note can be chosen to apply to *any* temperature sensor or to *one* particular sensor. In detail the matching process between two fields is as follows (we assume the fields match apart from possible labels):

- if the stick-e note field has no label then the fields match, even if the current context field is labelled. Thus the stick-e note field `<temperature> 63` would match any labelled temperature setting.
- if both have labels, then they match only if the labels are identical (labels *are* case-sensitive). Thus `<temperature> sensor1= 63` does not match `<temperature> sensor2= 63`.
- if the stick-e note field has a label, but the current context does not, then there is no match.

Clearly this facility is a first step towards structuring of context fields. An earlier implementation of the Matcher had a more advanced system of hierarchical labels (e.g. the label `person/engineer/Sue=`), but this has now been dropped. In the future we may have available general *context servers* which offer such structuring, and many other extra goodies besides.

Asymmetry of matching

Because of labels, the matching between a note and the current context is asymmetric, i.e. note A may match current context B, but if B is treated as a note it would not match A as the current context. Asymmetry can also arise in the matching of field values (though not in the current implementation). It arises, for example, if two values have a high matching-score only if the range of the first completely encompasses the range of the second.

Structural tags

So far we have described the fields within scontexts. Scontexts can also contain additional tags, which are called *structural tags*. Stick-e notes come in several types —

we discuss these later — for example:

- an information note: this is the default case, but it may be introduced by a `<note>` tag.
- (Todo do we keep or revise this concept?) a setting of the active tags, introduced by a `<activeTags>` tag.

A structural tag at the start of a note, e.g. a `<note>` tag. If scontexts are stored end-to-end within a file, rather than being within a database, then these structural tags also serve to show where one note ends and the next begins. Whenever an `<activeTags>` scontext is encountered it resets the properties of the matching process; it does not form part of the stick-e notes or the current context.

PJB decision: previous idea of subcontexts is abandoned; it covered several facilities (conversion rules, re-setting fields, match reports), but all need to be rethought when we go from Boolean to best-match.

Structural tags have reserved names and cannot be used for field names. They cannot have associated data values or labels. The full list of structural tags is currently: `<note>`, `<activeTags>` and `<MatcherMessage>` (an error or warning note from the Matcher: these are represented as scontexts and can be embedded in other output data — see later). There is also `<notes>` (at least for historical reasons -- covers a document collection); PJB thought: omit this for the time being; we may need to think about multiple document collections, each beginning with a `<notes>` tag and perhaps some ID for each collection, but not yet. Finally we need a way of resetting the switches on the call to the Matcher: if there is a sequence of current contexts supplied, then switches (e.g. field weights) can be reset before each context in the sequence; one approach is to have a `<CM−switches>` tag, the other is to have a SWITCHES attribute on any `<notes>` tag that begins a new current context. I suggest the latter. Since it is messy to keep adding reserved names, I suggest we reserve `<CM_...>` for future reserved tags.

The matching process

The matching process compares each of the stick-e notes with the current context. This is called a *matching pass*. Each comparison of the current context with a stick-e note is done independently; the output is derived from the best matches, and is sorted so that the highest scoring match comes first. Matching passes are repeated periodically (e.g. every second, every hour, whenever anything changes) depending on the nature of the application.

All parameters of the matching process are global: i.e. during a matching pass each stick-e note is matched in the same way. Thus the activeTags specification is the same for all matches in a matching pass.

To be precise, a match occurs if all active fields in the note match corresponding fields in the current context (remember that we are assuming the proactive case throughout this discussion). The asymmetry we mentioned earlier in matching individual field values carries over into the whole process. As an example if `<x>` and `<y>` are active fields, and we have the scontext (A) which is:

```
<x> 3
<y> 4
```

and a further scontext (B) which is:

```
<x> 3
```

then if (A) is the note and (B) the current context there is no match, since the active `<y>` field of the note is not matched. If, however, the roles are reversed there is a match, since the only active field in the note is `<x>`, and this is matched. One way of looking this is to regard each stick-e note as a query, whose active fields interrogate the current context.

In the interactive case the roles are reversed. (PJB comment: currently controlled by a parameter of the Matcher; you can have a pipeline of Matchers with some proactive and some interactive.)

Setting the active tags

The currently active tags are controlled by the most recent occurrence of an `<activeTags>` scontext. If no such setting has occurred, defaults apply. A sample `<activeTags>` scontext is

```
<activeTags>
  <location> ANY
  <temperature> 0..100
  <facing> Jones= ANY
```

This is matched against each active stick-e note as a first stage, and the fields that match (with a matching-score above some threshold) are the active ones; these are therefore the ones matched, as a second stage, against the current context to see if an overall match occurs. As a detail, in the first stage, the `<activeTags>` specification is treated as a current context that is matched interactively, not proactively, against the each of the stick-e notes, though this interactiveness only matters in asymmetric cases, e.g. when labels are present.

The default settings of the active tags are:

- for each stick-e note, the active fields are defined to be those whose *names* match correspondingly named fields in the current context. Thus if the current context has a `<pressure>` field, all `<pressure>` fields in notes are active. (Although, as we have said, the stick-e note drives the matching process in the proactive case, the current context has in this case an influence on how the match is performed.)

For each active field the normal matching rules apply: thus the field will match if the labels (if any) and values match too; if they do not the overall match will fail.

If an `<activeTags>` scontext is null, then it causes the above defaults to apply. Often an `<activeTags>` scontext is placed on the front of a set of notes as a suitable default ("this city tour is based on location and time"), but these defaults can be overridden by subsequent `<activeTags>` scontexts generated by the user/application (e.g. the user ask for an active tag to be switched off, and as a result the application would pass a new setting of `activeTags` to the Matcher). The active tags come into force when `<activeTags>` is parsed, which is prior to when matching takes place. Each setting overrides the previous one.

Closing tags

Closing tags (e.g. `</note>`) may optionally be included, but are ignored (unless they are wrong and give rise to an error). Nothing, apart from a possible comment, should come between a closing tag and the next tag. The following is an example:

```
<note> <y> 4
</y> </note>
```

At the end of an input file it is assumed that all tags are closed, i.e. nothing carries over to the next input file.

Nullity

A null stick-e note or a null current context will never match.

Error messages

Error messages from the Matcher are, by default, placed in the normal output stream. Each such message begins with `<MatcherMessage>` and ends with `</MatcherMessage>`. The messages usually come at the start of the output, since they arise during preliminary parsing.

Often the Matcher is used in a pipeline, where the output from one usage of the Matcher feeds into a second usage of the Matcher, and so on. If the first produces an error message, the second will encounter the message in its input. There is a simple (crude?) mechanism to deal with this: if the Matcher finds `<MatcherMessage>` in its input, it copies the entire message to its output, and otherwise ignores the message. Thus all error messages that arise in a pipeline make it to the final output.

THE SECOND STAGE: DELIVERY

Introduction

(PJB comment: previously we had matchReports and recipes, which allowed the Matcher to deliver a description of the results of the match, and allowed the application to pick out fields of the matchReport in order to display them. The new scheme described here is hopefully (a) simpler and (b) better geared to best-match retrieval.) The Matcher delivers to the application the stick-e notes that best match the current context; this applies in both proactive and interactive cases. Each stick-e note will have a numerical matching-score, which describes how well it matched, and the Matcher will sort the delivered notes, ordered in descending scores. To save bandwidth the application may set a threshold matching-score, thus stopping the Matcher from delivering any stick-e note with a lower score.

Details

In addition the Matcher needs to give information about the matching process. This information can be used by the application in deciding how to present information to the user, and it may also be useful for postprocessing and for debugging. It may also help the application to provide the user with facilities such as "Tell me the closest" (i.e. find, among the delivered stick-e notes, the ones that best match on location -- even though they might no have the best scores for overall match).

I suggest that matching information is conveyed by attributes, added by the Matcher to the `<note>` tag, and also to the tags for each of the fields that is involved in the matching. There are two such attributes: SCORE and AGAINST. The AGAINST tag is used for fields and gives the value of the corresponding field of the current context against which the stick-e note field was matched. (In many cases the AGAINST field could be deduced by the application, but it is useful: (a) when there has been pre- or post-processing; (b) when there have been time delays from when the original request was issued; (c) for debugging. An example of a matched stick-e note to which SCORE and AGAINST attributes have been added is:

```
<note SCORE="1.4">
   <author> ...
   <location score="1.6"
     AGAINST="23,45"> 23, 44..55
   <temperature score=".8"
     AGAINST="16"> 20
   <body> ...
   <openingHours> ...
```

CHANGE

(PJB comment: this issue has now come to the fore; the cited document represents old ideas; we are now working on new ones.) For information on how to detect change, e.g. changes in stick-e notes or change between one triggering and the next, see the separate document entitled "change". http://www.dcs.ex.ac.uk
/˜pjbrown/papers/change.html

In particular this explains the `<uniqueID>` field that is automatically added to many contexts.

RUNNING THE MATCHER

We now switch to more practical matters. We here give a brief explanation of how to run the Matcher as an ordinary programme. We here use the Unix command syntax. It is executed using the command:

`matcher` *switches arguments*

There must be at least two arguments: the last is treated as the present context, and all the preceding ones as stick-e notes. (PJB comment: this is crude and a later implementation tried something better, using "stages"; it was more pipeline oriented.) Each argument may be a literal or a filename or a URL; switches can be used to specify which of these applies, but if there are no switches, the Matcher makes an intelligent guess. Each argument must represent a single scontext, or a sequence of scontexts written end to end. All or part of an argument may be devoted to setting the the active tags.

There may be switches at the start or between arguments. The allowable switches are:

–a The default activeTags are the tags used in the current context. (This is the default anyway, but setting this switch suppresses a warning message saying that the default has been applied.)

–c means suppress SGML comments in the output.

–d means subsequent arguments are to be given the default interpretation (i.e. the Matcher's intelligent guess whether it is a file, URL, etc).

–f means subsequent arguments are filenames.

–l means subsequent arguments are literal strings.

–r means apply matching in the reverse direction (interactive instead of the default, proactive).

–s S means select scoring algorithms — see discussion below.

–u means subsequent arguments are URLs.

–v means verbose.

–w S
means set field-weights — see discussion below.

The Matcher can be accessed as a web server as:

`http://triton.ex.ac.uk/cgi-bin/cgiwrap/pjbrown/matcher`

This server has the notes as the first argument and the current context as the second; it returns its output as plain text.

LIST OF ATTRIBUTES

Some authors of SGML-based notations use attributes extensively, some very little. Thus in our case we could have provided a host of possible attributes for the `<note>` tag, e.g. AUTHOR, TITLE, DATE, etc., or we could have taken the view that if this information is needed it should be expressed as extra fields within the `<note>`, e.g. an `<author>` field. Overall we currently favour the latter: a consequence of the approach that the Matcher has no prior knowledge of what field names are used is that the user/author can add new fields painlessly. In a previous version of the Matcher attributes were used on fields to indicate the type of the value, e.g. whether a location was GPS co-ordinates, OS grid co-ordinates, etc. We may need to add such attributes in the future, but will do without them at present. We will assume therefore that the data type of all values is manifest from the value itself. Below we describe the few attributes that we do use.

Following the normal SGML rules, a tag can have a set of possible attributes associated with it. All the attributes used here are optional (i.e. "implied" in SGML parlance): they can occur 0 or 1 time on each tag. If a tag has two separate attributes, e.g. a SCORE attribute and an AGAINST attribute, these can occur in either order.

The following is a list of attributes that are used:

(1) The SCORE="*N*" attribute can occur on the `<notes>` tag, where it gives the document score, *N*. *N* must be a real number within the range allowed for scores. This attribute can also occur on any field tag used in the matching process. The attribute is inserted by the Matcher and added to the relevant tags in each document it retrieves and delivers to the user. The attribute can also occur within documents from the document collection that are *input* to the Matcher (after all, the output from one instantiation of the Matcher can be the input to a subsequent one, e.g. when a proactive retrieval is followed by an interactive one); in this case the value *N* is used as the initial value of the score. (We still need to decide how initial values are used by the scoring algorithms; maybe the default algorithms will multiply the newly calculated score by the initial score; algorithms plugged in by the user can use the initial score as they wish — it

therefore should be accessible.) The SCORE attribute should not be used within the current context (there is an error message if this rule is broken).

(2) The AGAINST="*V*" attribute is used in connection with the SCORE attribute when the SCORE attribute occurs on a field name rather than on the `<note>` tag. *V* gives the value of the field in the current context that was matched against the current field to yield the SCORE. The AGAINST tag is ignored on input; it should not occur at all in the current context.

(3) THE SWITCHES="*S*" attribute can be attached to any `<note>` tag that begins a current context. *S* follows exactly the same syntax as the switches on the Unix command to launch the Matcher (see later); e.g.

```
note SWITCHES="-w at:4 -r">
  ...
```

There may be some restrictions on what switches can be reset dynamically.

SOME EXAMPLES

(PJB comment: these date from Boolean matching and need revising.) Many people learn better from examples than from reading lots of text, so we finish by giving a series of examples. We do not here go into the exact scoring mechanisms used for matches, and just assume the matched stick-e notes are output in the order they occur.

Matching of stick-e notes

The first set of examples assumes that the stick-e notes are:

```
<note>
<title> note 1
<at> 1..6, 7..12
<body> Cathedral

<note>
<title> note 2
<at> 6..9, 12..14
<body> Chapter House

<note>
<title> note 3
<at> 1..100, 0..200
<temperature> 70..
  <!-- Fahrenheit used -->
<body> Swim in the river
```

(Obviously this is an abbreviated form of what real notes are likely to be.) It is a useful convention, followed above, to put `<title>` fields in each stick-e note, which, though not designed to take part in the matching process, help keep tabs on what is going on.

We first assume the current context:

```
<!-- example n1 -->
<at> 1, 10
```

If we assume the default activeTags are in place, then the `<at>` field is the only active one; this applies to all three notes, since, in each case, it is the only tag in both the note and the current context. Given this, then two of the above three notes are matched. (The location on the current context comes with the range specified on two of the stick-e notes, but is outside on the third case; we assume the former are treated as matches.) A common form of post-processor is one that extracts from the stick-e notes the fields that were *not* involved in the matching: these fields are likely to be information fields that are of interest to the user. If we assume that there is such a post-processor, then the output produced is:

```
<title> note 1
<body> Cathedral
<title> note 3
<temperature> 70..
<!-- Fahrenheit used -->
<body> Swim in the river
```

Actually it is best practice to set explicitly the activeTags. We will do this, and try a different current context:

```
<!-- example n2 -->
<activeTags>
<at> ANY
</activeTags>
<at> 6, 12
```

In this example, the location is within the range specified on each of the three stick-e notes, and we therefore assume that all count as matches. The output will be:

```
<title> note 1
<body> Cathedral
<title> note 2
<body> Chapter House
<title> note 3
<temperature> 70..
<!-- Fahrenheit used -->
<body> Swim in the river
```

We will now introduce a current context with two fields, and make both active:

```
<!-- example n3 -->
<activeTags>
<at> ANY
<temperature> ANY
</activeTags>
<at> ANY
<temperature> 32
```

This will produce the output

```
<title> note 1
<body>Cathedral
<title> note 2
<body>Chapter House
```

The note that requires a temperature of 70 or above is, we assume, not retrieved, as it will get a low matching-score with the current temperature at 32. If, however, we made `<temperature>` the *only* active tag by means of:

```
<!-- example n4 -->
<activeTags>
  <temperature> ANY
</activeTags>
<at> ANY
<temperature> 32
```

then, since the note with a `<temperature>` field gets a very low matching-score, and since the other two notes are not active since they do not contain an active field, nothing will be retrieved (unless, of course, the application had an extremely low threshold score for retrieval).

Finally if we revisit example n1 above and assume all the fields are output we might see:

```
<note SCORE="1.8">
<title> note 1
<at SCORE="1.8"
  AGAINST="1, 10"> 1..6, 7..12
<body> Cathedral

<note SCORE="1.7">
<title> note 3
<at SCORE="1.7"
 AGAINST="1, 10"> 1..100, 0..200
<temperature> 70..
  <!-- Fahrenheit used -->
<body> Swim in the river
```

(We assume the first note gets a higher matching-score because it covers a smaller area, and because one of the user's location co-ordinates is central to this area.)

ISSUES FOR DISCUSSION

Pre-loading the stick-e notes

In the current implementation, the Matcher is driven by a command line whose arguments specify all the data to be used (stick-e notes, current context, switches, etc.), though you could have a test harness which, e.g., invoked the Matcher once a minute. In the current implementation, if you restart the Matcher you have to re-load all the stick-e notes again; this is very limiting, especially in the test harness case. I think the new implementation should allow users to create a process consisting of the Matcher with a given set of stick-e notes loaded, and then call this process, supplying a different current context with each call. Alternatively we could have a web server, which had a set of stick-e notes pre-loaded; each user would supply their current context, and retrieve the notes that were relevant to them.

Following on from this, if you have pre-loaded stick-e notes, it would be desirable to allow updates on-the-fly, without a complete re-load. For example we might want to add new stick-e notes (relating, say, to a sudden traffic problem) and/or to delete existing ones. Likewise we might want to change the switches, e.g. to be interactive rather than proactive.

Default scoring algorithms for field values

In the short/medium term we need to have some default scoring algorithms when matching two field values, which we here call the *query* field value and the *target* field value. Some suggestions are below. We assume numeric values are ranges; we also refer to the centre of the range; if the value is a single point the range and the centre of the range will be one and the same. The assumption that values are ranges means we are assuming 2D areas are rectangles. The following suggestions cover numeric values:

- The centres of the two ranges should be as close as possible. Perhaps the (inverse of the) square of the distance between the two should be used in scoring.
- A small range on the target is better than a large one (Exeter Cathedral is better than Exeter, which is better than Devon, ...). (Hence a match of `ANY` would get a low matching-score.)
- It is best if the query range lies totally within the target range.

The following suggestions cover string matching (testing if the target value contains the query value):

- The more occurrences of the query value in the target the better.
- Matches early on and better than matches later on.
- Maybe "near misses" should count for something, e.g. "stick-e" and "sticke".

The following suggestions cover value-tuples:

- The matching-score for a value-tuple (which may involve mixed data-types such as `23,YES,10`) should be the mean (arithmetic or geometric?) of the scores of the individual values.

Default algorithms for accumulating note matching-scores from its constituent field scores

The algorithm for deriving an overall score for a note will almost certainly be a weighted mean of the field scores, with the convention that a compulsory field that does not match or gets a score of 0 signals a complete non-match and therefore sets the overall matching-score to 0 (this happens anyway if you use geometric means). It seems obvious that by default all the weights of individual fields should be equal: only the application (and/or pre- and post-processors) will have the knowledge and experience to change the defaults (e.g. by noting rarely matched fields and weighting them more highly). It also seems that N+1 fields matching, each with an average matching-score, is better than N fields matching, each with this same average matching-score. Perhaps the best approach is a weighted arithmetic mean where, instead of diving by N at the end, you divide by (say) N-1/2.

Compulsory and optional fields

In various implementations I have continually wavered between (a) having both compulsory and optional fields, and (b) just having compulsory fields. A key property of an optional field is that if it fails to match (e.g. does not exist in the target or gets a matching-score of 0) it does not knock out the whole match. If you only have (b) you cannot completely simulate (a) using scoring algorithms, and default values for missing fields (e.g. if you have no location sensor you set your location to ANY and give location a low field-weight), but you can go some way. With this in mind do we adopt approach (b) for the short term?

Setting parameters of the matching process

Parameters of the matching process can either be set globally by switches on the command that launches the Matcher, or they can be set within the data. The only example of the latter is currently the `<activeTags>` facility. We need ways of setting the scoring algorithms and the field-weights. I think we are moving towards the algorithms being pieces of Java code that are incorporated before the Matcher starts; however the user may supply several alternative algorithms and may wish to place a switch-value within the data to select dynamically the algorithm to be used for each set of test data. Field-weights can more easily be supplied as dynamic values, but it may also be desirable to supply them on the command-line.

Can I suggest the fully overall scheme:

- we define a UNIX-style command-line interface for setting parameters for the matching process.
- we allow a declaration of form:

  ```
  <parameters> XXX
  ```

  within the data (e.g. before a new setting of the current context) and the syntax of XXX exactly follows the syntax of the command line. (I do not think it matters too much if the notation follows a typically terse and somewhat unfriendly UNIX style.)

An example of a parameter setting might be

```
<parameters> –w at:2;temperature:.6 –s at:4 –r
```

This might (a) set the field-weight of `<at>` fields to 2 and `<temperature>` fields to .6; (b) set the scoring algorithm for `<at>` fields to algorithm number 4 of the ones supplied; (c) set the –r parameter so that subsequent matches were done in reverse (i.e. interactively rather than proactively).